NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

SYMMETRIC LINK KEY MANAGEMENT FOR SECURE NEIGHBOR DISCOVERY IN A DECENTRALIZED WIRELESS SENSOR NETWORK

by

Kelvin T. Chew

September 2017

Thesis Advisor:	Preetha Thulasiraman
Second Reader:	Murali Tummala

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704–0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2017	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE SYMMETRIC LINK KEY MANAGEMENT FOR SECURE NEIGHBOR DISCOVERY IN A DECENTRALIZED WIRELESS SENSOR NETWORK			**5. FUNDING NUMBERS** **W7B46**
6. AUTHOR(S) Kelvin T. Chew			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Marine Corps Systems Command and Naval Research Program			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB number ____N/A____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Wireless sensor networks provide a low-signature communications system that can be used for a wide variety of military applications. These networks are vulnerable to intrusion, however, and must balance security with performance and longevity. The neighbor discovery process is vital for nodes to maintain network connectivity but introduces security vulnerabilities; therefore, a lightweight security protocol is necessary to prevent unauthorized nodes from accessing network data and resources. In this thesis, we focus on the management of encryption keys in a resource-limited, peer-to-peer, decentralized network. Existing protocols for securing the neighbor discovery process use public key encryption, which is too computationally expensive for low-powered, resource-constrained IEEE 802.15.4-enabled devices. We therefore develop a key management scheme that modifies the Neighbor Discovery Protocol (NDP) and Secure Neighbor Discovery (SEND) protocol and implements the Diffie-Hellman key exchange algorithm for symmetric key management. We simulate our scheme in MATLAB to demonstrate its effectiveness in securing the neighbor discovery protocol while providing energy efficiency, key security, and error resistance.

14. SUBJECT TERMS wireless sensor network, 6LOWPAN, key management, neighbor discovery, symmetric cryptography, identity-based cryptography, Diffie-Hellman key exchange, cyber			**15. NUMBER OF PAGES** 103
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540–01-280-5500 Standard Form 298 (Rev. 2–89) Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

SYMMETRIC LINK KEY MANAGEMENT FOR SECURE NEIGHBOR DISCOVERY IN A DECENTRALIZED WIRELESS SENSOR NETWORK

Kelvin T. Chew
Captain, United States Marine Corps
B.S., Virginia Tech, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2017

Approved by: Preetha Thulasiraman, Ph.D.
Thesis Advisor

Murali Tummala, Ph.D.
Second Reader

R. Clark Robertson, Ph.D.
Chair, Department of Electrical Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Wireless sensor networks provide a low-signature communications system that can be used for a wide variety of military applications. These networks are vulnerable to intrusion, however, and must balance security with performance and longevity. The neighbor discovery process is vital for nodes to maintain network connectivity but introduces security vulnerabilities; therefore, a lightweight security protocol is necessary to prevent unauthorized nodes from accessing network data and resources. In this thesis, we focus on the management of encryption keys in a resource-limited, peer-to-peer, decentralized network. Existing protocols for securing the neighbor discovery process use public key encryption, which is too computationally expensive for low-powered, resource-constrained IEEE 802.15.4-enabled devices. We therefore develop a key management scheme that modifies the Neighbor Discovery Protocol (NDP) and Secure Neighbor Discovery (SEND) protocol and implements the Diffie-Hellman key exchange algorithm for symmetric key management. We simulate our scheme in MATLAB to demonstrate its effectiveness in securing the neighbor discovery protocol while providing energy efficiency, key security, and error resistance.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

6LoWPAN	IPv6 over low-rate wireless personal area network
AES	Advanced Encryption Standard
BS	base station
C2	command and control
CA	certificate authority
CGA	cryptographically generated address
DH	Diffie-Hellman
DOD	Department of Defense
ECB	Electronic Codebook Mode
IBC	identity-based cryptography
ICMPv6	Internet Control Message Protocol version 6
ID	identifier
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPSec	IP Security
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
KDC	key distribution center
LR-WPAN	low-rate wireless personal area network
MAGTF	Marine Corps Air Ground Task Force
MCEN	Marine Corps Enterprise Network
MD5	Message Digest 5
MTU	maximum transmission unit
NDP	Neighbor Discovery Protocol
NWG	Network Working Group
PKC	public-key cryptography
PKE	public-key encryption
PKG	public key generator
PKCS	Public Key Cryptography Standards

PKI	public-key infrastructure
RF	radio frequency
RSA	Rivest-Shamir-Adleman
SEND	Secure Neighbor Discovery
SHA	Secure Hash Algorithm
TRSS	Tactical Remote Sensor System
USMC	United States Marine Corps
WSN	wireless sensor network

ACKNOWLEDGMENTS

I would like to first thank my wife, Pamela, for her support and understanding while I pursued my graduate education. And to our dogs, Monknet and Ginger, thank you for keeping me company during all those late nights.

I would like to thank my thesis advisor, Professor Preetha Thulasiraman, for her support, guidance, and patience while we developed and executed the idea for this thesis.

To my fellow students and the faculty of the Electrical Engineering Department, thank you for making this such an enjoyable experience.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

As a worldwide expeditionary force in the 21st century, the United States Marine Corps (USMC) has prioritized the protection of command and control (C2) systems and information networks [1]. These vital assets enable a commander to obtain and process information to make timely and informed decisions on the battlefield [1]. As our adversaries expand their network capabilities, the Marine Corps Air Ground Task Force (MAGTF) must adapt to a highly "contested-network environment" to exploit its advantages while denying the enemy those same advantages [1]. The key objectives are to reduce friendly electromagnetic signatures and harden our networks against degradation or compromise [1]. Ultimately, the MAGTF must leverage existing communications assets to minimize signature and defend networks while providing adequate C2 capability [1].

One such low-signature system in the USMC arsenal is the Tactical Remote Sensor System (TRSS), which is detailed in [2]. The TRSS supports the MAGTF by providing a continuous surveillance system that monitors activity in a chosen area [2]. Remote sensor operations are an economical and low-risk approach to expand the commander's ability to collect information by providing general surveillance, early warning, and target acquisition [2]. Furthermore, this system can be integrated into the MAGTF network infrastructure for real-time, remote monitoring of the battle space [2]. When used properly, the TRSS is very difficult to detect due to its small electromagnetic footprint and built-in electronic countermeasures [2].

The potential for military applications of wireless sensor networking vastly expands beyond surveillance and targeting. Sensor network applications also include chemical, biological, radiological, nuclear, and explosive detection, ranging, imaging, and acoustic tracking [3]. As a force protection measure, personnel can also wear sensor devices that track vital biometrics and unit locations [3]. Sensor network operations clearly contribute to the USMC's priority of developing low-signature systems, as outlined in [1].

A. WIRELESS SENSOR NETWORKS

Wireless sensor networks (WSN) are comprised of specialized sensors and actuators and linked together via a wireless communications infrastructure [4]. Wireless sensor networks can either monitor physical or environmental conditions by passing data from a remote sensor to a central location or control remote systems by passing commands from a central location to a remote actuator [4]. The applications for sensor networks, which are the focus of this thesis, are broadly divided into two categories: remote monitoring and mobile object location tracking [4]. Remote monitoring applications periodically measure environmental conditions, while tracking applications generally provide real-time data updates of the target being tracked [4].

These networks consist of multiple nodes wirelessly linked together and designed for a specific function [4]. Sensor nodes, or end nodes, sense and collect data at the remote location [4]. Meanwhile, router nodes relay data within the network, and sink nodes, or base stations (BS), act as a gateway to exchange data with external networks [4]. These nodes tend to be small, low-power devices with low data rates and short transmission ranges [4]. They are battery-powered and communicate with each other via a radio transceiver [4].

Although WSN support many useful applications at a relatively low cost, their implementation introduces several unique challenges. First, the low power capacity of each node places a high premium on the energy efficiency of the hardware, data processing methods, routing algorithms, and security protocols [4]. Second, the limited computational capability of the nodes prevents the use of complex, highly iterative processes [4]. Finally, the wireless nature of WSN presents the problem of security where we must protect the network and its data from passive and active attacks or intrusions [4].

B. WIRELESS SENSOR NETWORK STANDARDS

The unique operational constraints of these WSN devices prompted the Institute of Electrical and Electronics Engineers (IEEE) to develop the IEEE 802.15.4 standard. This standard provided wireless connectivity among these devices with a scalable data rate depending on the application [5]. With the advent of Internet Protocol version 6

(IPv6), the IPv6 over Low-power Wireless Personal Area Network (6LoWPAN) protocol was developed to allow IPv6 data to transit IEEE 802.15.4 networks.

1. IEEE 802.15.4

In 2003, the IEEE developed the IEEE 802.15.4 standard for very low-cost, lower-power communications over low-rate personal area networks (LR-WPAN) [5]. The objectives of the IEEE 802.15.4 standard were to develop a simple and flexible protocol that simplified network installation, reduced cost, and provided efficient and reliable data transfer [5]. Since WPAN transmit information over short distances, the standard requires minimal infrastructure and can accommodate the small, lower-power devices often used in WSN applications [5].

This standard provides for basic security services to include data confidentiality, data authenticity, and replay protection [5]. IEEE 802.15.4 designates Advanced Encryption Standard (AES)-128 as the block cipher, using 128-bit symmetric keys for encryption [5]. The cryptographic security mechanism uses either a group key that is shared among a group of devices or a link key which is shared only between two devices [5]. When applied in accordance with federal information processing standards in [6] and [7], AES-128 fulfills the Department of Defense (DOD) requirements for computer security.

2. IPv6 over Low-Power Wireless Personal Area Networks

IPv6 was developed in the late 1990s to address the unforeseen shortfalls of IP version 4 (IPv4). IPv6 expanded the IP address size from 32 bits to 128 bits, simplified the header format, and improved the encoding of extensions and options to support data authentication, integrity, and confidentiality [8]; however, the limitations of IEEE 802.15.4 required the Internet Engineering Task Force (IETF) to develop IPv6 over Low-power WPAN (6LoWPAN) to enable the transmission of IPv6 packets over IEEE 802.15.4 enabled devices [9]. The 6LoWPAN protocol specifically provides for packet fragmentation and header compression necessary to make IPv6 practical on IEEE 802.15.4 networks [9].

Packet fragmentation is necessary due to the significant disparity between the standard IPv6 and IEEE 802.15.4 packet size. The maximum transmission unit (MTU) for IPv6 packets is 1280 octets, while the MTU for IEEE 802.15.4 packets is 127 octets [9]. When factoring in a maximum frame overhead of 25 octets and security overhead of 21 octets, the IEEE 802.15.4 only has 81 octets of practical payload capacity [9]; therefore, the 6LoWPAN protocol defines an adaptation layer that performs packet fragmentation and reassembly at the data link layer to allow transmission of larger IPv6 packets within the MTU constraints of IEEE 802.15.4 [9]. The 6LoWPAN adaption layer is shown alongside the standard IP stack in Figure 1.

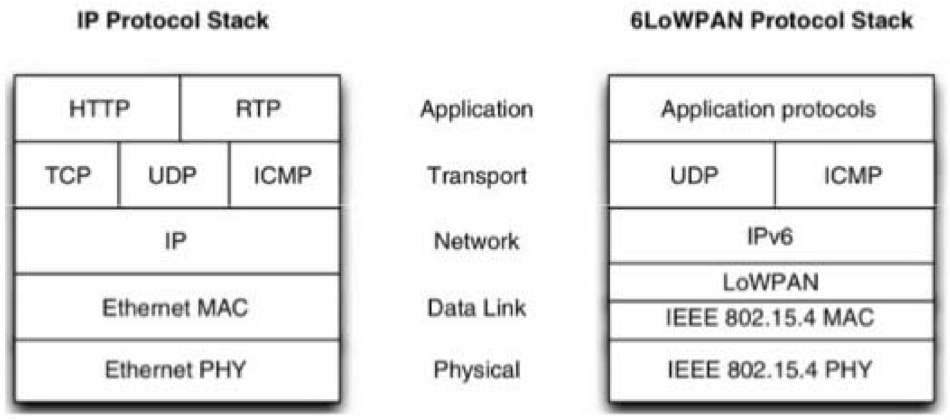

Figure 1. IP and 6LoWPAN Protocol Stacks in TCP/IP Model. Source: [10].

In addition to fragmentation, the 6LoWPAN protocol must perform header compression to account for the larger IPv6 header. Since the IPv6 header is 40 octets long, a normal IEEE 802.15.4 packet with a payload capacity of 81 octets only has 41 octets remaining for application data [9]; therefore, header compression is necessary to reduce the overhead and maximize useful capacity. In the 6LoWPAN protocol, the header can potentially be compressed from 40 octets down to two octets by compressing common header values inherent to 6LoWPAN networks or by inferring values from other sources within the packet [9].

C. RESEARCH MOTIVATIONS AND OBJECTIVES

As recently as July 2017, the USMC published updated policy [11] mandating the use of public key infrastructure (PKI) on all Marine Corps systems and devices accessing Marine Corps Enterprise Network (MCEN) resources. Encryption is a vital technique in providing confidentiality and authentication for transmitted data [12]. Consequently, PKI is an essential component of public key cryptography (PKC) that provides authentication for public keys used in the network [13]. By directing the use of PKI across all network systems, the USMC has taken an important step in hardening the defenses of its most widely used network.

Although the MCEN is a garrison network infrastructure, it is even more important that we extend these defensive measures into a tactical network environment in the spirit of USMC priorities discussed in [1]. The USMC's aging TRSS relies primarily on stealth and tamper alarms for physical security, but it does not possess any security measures, such as encryption, to protect its radio frequency (RF) data transmissions [2]. This security gap leaves the TRSS vulnerable to malicious systems acting as legitimate nodes and infiltrating the network's data and resources; therefore, we must leverage the IEEE 802.15.4 and 6LoWPAN protocols to provide the USMC with a wireless sensor system that provides the same sensing and tracking capability while ensuring continuous security of its network.

The objective of this research is to develop a key management system to provide secure neighbor discovery in a decentralized, resource-limited, peer-to-peer wireless sensor network. We focus on end-device communication in a decentralized network without distributed routers to perform higher level functions. First, we study the most efficient encryption and key generation methods available such that the energy constraints of a WSN are considered. Second, we aim to minimize secret key exposure, which impacts the security of the network, specifically if a node and its stored keys are compromised. Finally, we design a scheme that does not require a central hub for key management but instead is performed solely by the end nodes.

D. THESIS CONTRIBUTIONS

To achieve these objectives, we develop a symmetric link key management scheme that implements the Diffie-Hellman key exchange algorithm to decrease computational overhead while protecting the network against secret key exposure.

The contributions of this thesis are:

- Development of a symmetric link key management scheme for WSNs that generates and maintains link keys without a centralized node.
- Simulation of the key management scheme to validate its effectiveness in finding one-hop neighbors and generating and maintaining link keys while staying within the operating constraints.
- Measurement of the performance of the key management scheme for message efficiency, key distribution, energy consumption, and error resistance.

In our review of prior literature, we found no other research that uses the Diffie-Hellman key exchange algorithm to generate and distribute symmetric keys for securing the neighbor discovery process in decentralized wireless sensor networks.

E. THESIS ORGANIZATION

The remainder of this thesis is organized into five chapters. In Chapter II, we discuss relevant background information and previous research on neighbor discovery security protocols. Our proposed scheme for symmetric key distribution for secure neighbor discovery is detailed in Chapter III. Next, our experimental design is described in Chapter IV, to include the simulation parameters, program structure, and performance metrics. In Chapter V, we present the simulation results and discuss the significant implications for each of the performance metrics. Finally, we draw conclusions and propose related topics for future work in Chapter VI. All of the MATLAB code used for our simulations is included in the Appendix.

II. BACKGROUND AND RELATED WORK

In this chapter, we discuss the background that frames our work in symmetric key management for secure neighbor discovery in decentralized WSN. We begin with an overview of the Neighbor Discovery Protocol (NDP) and Secure Neighbor Discovery (SEND), which serve as the basis for our research. Then, we review existing research in key management for secure neighbor discovery.

A. NEIGHBOR DISCOVERY PROTOCOL

All WSN nodes perform neighbor discovery to find and track the active nodes within their transmission range and detect changes to node addresses. In 1998, the Network Working Group (NWG) developed the IPv6 NDP [14], which is the basis for our key management scheme. The NDP defines the supported link types, addressing methods, neighbor discovery message formats, and functions of the messages transmitted between neighbor nodes [14].

This protocol supports addressing for several types of links, but we focus on the multicast and point-to-point links. Since IPv6 does not support broadcast [8], a multicast link is used to support data transmission to all nodes within range [14]. By sending a message to an all-nodes multicast address, a node can effectively "broadcast" that message to all known and unknown nodes within transmission range. The point-to-point link connects exactly two nodes together and is serviced with a unicast message sent from one node directly to the link-local address of the other node [14].

The NDP defines five message formats to include the router solicitation, router advertisement, neighbor solicitation, neighbor advertisement, and redirect message formats [14]. Since we focus on one-hop, peer-to-peer communication without distributed routers, we concentrate only on the two neighbor message formats in this thesis.

Neighbor solicitations are sent to request the link-layer address of the target node while providing the link-layer address of the sending node [14]. These messages are multicast when a node is discovering a new node's address or resolving a known address

[14]. Otherwise, neighbor solicitations are unicast when a node is verifying the reachability of a known neighbor [14]. Neighbor advertisements are sent as unicast responses to acknowledge solicitations [14].

The nodes create and maintain associations with their one-hop neighbors via the neighbor discovery messages. When a node is first activated and subsequently at a pre-defined frequency, it initiates the discovery process by sending out neighbor solicitations to all nodes within transmission range [14]. The solicitation message is sent to the all-nodes multicast address and includes the sender's own address [14]. When an in-range node receives the solicitation, it verifies the target multicast address from the message against its own address [14]. It then creates a neighbor cache entry for the sending node and responds with a unicast neighbor advertisement back to the sender's address [14]. The sending node then receives the advertisement and updates its neighbor cache to complete the neighbor association [14]. Nodes may also send unicast neighbor solicitations to addresses of *known* neighbors to verify that they are still active and that the stored address is still correct [14].

B. SECURE NEIGHBOR DISCOVERY

Although the NDP provides a simple but effective procedure for creating and maintaining neighbor associations in a WSN, the protocol must be secured to protect the network from attack or intrusion by malicious nodes. The original NDP specification [14] directed the use of IP Security (IPSec) to secure neighbor discovery messages [15]. The IPSec architecture provides robust security for host-to-host communications at the network layer; however, IPSec requires a large amount of resources, which makes it infeasible for use with WSN devices [16]. Additionally, IPSec experiences problems with bootstrapping in the NDP autoconfiguration process, so it is not suitable for use in neighbor discovery [15]. In 2005, the NWG developed the SEND protocol [15] to protect the NDP, especially over the wireless medium.

1. Neighbor Discovery Message Options

The SEND protocol protects the NDP by introducing a set of options appended to all neighbor discovery messages [15]. The new options are the Cryptographically

Generated Address (CGA) option, the Rivert-Shamir-Adleman (RSA) Signature option, and the Timestamp and Nonce options [15].

The CGA option is used to authenticate the sender of the message. The option contains the sender's public key and associated parameters that are used to generate the sender's CGA using the Secure Hash Algorithm 1 (SHA-1) [15], [17]. The message is then accepted by the recipient only if the message source address matches the CGA [15]. The primary disadvantage of the algorithm for CGA generation is its computational cost [18]. The mechanics of the algorithm, especially at higher security-level values, can require numerous SHA-1 computations on a single packet to meet the hash specification [18]. These iterations increase the processing time and energy consumption for each neighbor discovery message sent and received.

The RSA Signature option uses public key signatures to ensure message integrity and sender authentication [15]. Using the RSA algorithm and SHA-1 hash function, the sender computes its Public Key Cryptography Standards (PKCS) #1 digital signature with its private key over a number of message parameters and appends it to the end of the message [15]. The receiver verifies the sender's digital signature, again using the RSA algorithm and SHA-1 hash function [15]. The message is accepted only if the calculated signature matches the received signature [15]. RSA is based on PKC; however, PKC is generally considered too computationally costly for low-power devices [13].

In addition, the use of RSA-based digital signatures also requires public key authentication. Public keys must be authenticated to ensure that the public key contained in the neighbor discovery message belongs to the sending node. Public-key infrastructure is the mechanism that authenticates public keys by associating each public key to its respective node via a public-key certificate [13]. These certificates are generated and issued by a Certificate Authority (CA) [13]. In a WSN, a router or sink node with higher processing and power capacity are required to perform the functions of the CA. Without these router or sink nodes, the use of certificates to authenticate public keys is not feasible in a decentralized WSN.

The Timestamp and Nonce options are used to protect against replay attacks [15]. More specifically, the Timestamp is an integer time value that ensures unsolicited advertisements have not been replayed [15]. The receiver accepts a message only if the Timestamp does not exceed a certain “delta” time from the current time [15]. Meanwhile, the Nonce is a random number selected by the sender that protects against replay of solicited advertisements [15]. The receiver accepts an advertisement only if the Nonce of the advertisement matches the Nonce of the corresponding solicitation [15].

2. Further Disadvantages of SEND

The SEND protocol does not define a standard for message encryption and instead relies on IPSec as directed in NDP for message confidentiality. The resource cost associated with IPSec makes it impractical for use in our WSN devices. Consequently, all neighbor discovery messages are sent in the clear, allowing attackers to read node addresses, keys, and other vital data. Attackers can also interpolate packet traffic patterns and gain information about the network.

A study measuring the energy consumption of symmetric and asymmetric cryptographic algorithms was performed in [19]. The comparison between the energy cost to perform the RSA digital signature algorithm and AES encryption and decryption on a single 127-byte packet is shown in Table 1.

Table 1. Energy Consumption in Joules for RSA and AES. Adapted from [19] and [20].

Algorithm	**Key Setup**	**Sign/Encrypt**	**Verify/Decrypt**	**Total Energy**
RSA-1024	270.13 mJ	546.5 mJ	15.97 mJ	832.6 mJ
AES-ECB-128	7.87 µJ	205.74 µJ	316.23 µJ	529.8 µJ

Comparing total energy requirements for AES and RSA, we estimate that two neighboring nodes can perform the AES encryption and decryption of over 1,500 packets for every one RSA digital signature that is signed and verified; therefore, symmetric key encryption offers a significant advantage in energy efficiency, which is crucial to the longevity of the decentralized WSN.

Given the above-mentioned disadvantages, the current SEND protocol is insufficient to address the security concerns of decentralized WSNs employing low-power devices.

C. APPROACHES TO KEY MANAGEMENT IN NEIGHBOR DISCOVERY

Recent literature reveals several novel ideas for key management to support the security of neighbor discovery. These schemes predominantly apply only to centralized WSN that feature at least one higher function node to perform key management functions. In this section, we review some of these schemes from which we draw concepts to design our proposed scheme.

1. Group Key Establishment for Secure Multicast Communication

In [21], the authors devised a protocol for establishing group keys for multicast communications between an identified set of sensor nodes. First, an initiator node identifies a set of sensor nodes to create the multicast group [21]. The initiator then broadcasts a message along with its digital signature to initiate the group key establishment [21]. Next, each receiving node verifies the signature and computes a unique value using its private key with other parameters and returns the value to the initiator [21]. After receiving responses from all active nodes in the group, the initiator encodes all the unique values, generates a group key from the encoded values, and sends a multicast message with digital signature back to the group containing the encoded values and a hash of the group key [21]. Finally, each node computes the group key from the encoded values and verifies the key against the hash [21].

There are several issues that prevent the use of this scheme in a decentralized WSN. First, the scheme relies on public key encryption (PKE) and digital signature and, as discussed previously, the required computation and processing are too costly for the low-power devices in the network. Second, group keys are generated from the inputs of a select group of nodes. The addition of even a single node to the group renders the key useless, requiring a new key to be generated. Finally, the scheme requires each node to have complete knowledge of the network topology to form its groups; therefore, this scheme likely would not be useful in neighbor discovery.

We favor the idea of generating keys without the use of a trusted key distribution center (KDC). Our scheme features a mechanism for direct key establishment between nodes while allowing for flexibility when nodes become active or inactive. Finally, keys should be locally computed by the nodes rather than transmitted.

2. Network Admission Control Based on Symmetric Key Mechanisms

A network admission control scheme for 6LoWPANs using symmetric key encryption was presented in [22]. The scheme emulates the secure neighbor discovery process by detecting nearby nodes and performing node authentication and authorization to join the network. In this scheme, each node is preloaded with a unique symmetric key shared with the border router [22]. When detected by the border router, the node performs a one-way challenge authentication with the router [22]. With their shared key, the node can decrypt the challenge message containing the secret global network key that is used for communication with other nodes in the network [22].

This scheme also poses issues with our decentralized network environment. In a decentralized WSN, there is no border router to perform key exchange. Also, we wish to avoid the transmission of keys even if they are encrypted. Most importantly, the use of one global network key for node-to-node communication creates a single point of failure where the entire network's security can be compromised [13]. Nevertheless, this scheme presents the efficiency advantage of symmetric key cryptography and the idea of incorporating challenge authentication to neighbor discovery.

3. Identity-Based Cryptography

Identity-based cryptography (IBC) was introduced in [23] and allows for a node's public key to be generated from a node identifier (ID) [13]. In this system, every node is preloaded with an ID and private key [13]. Communicating nodes exchange only their node IDs which are then used to locally generate the appropriate public keys for communication [13]. Identity-based cryptography eliminates the need for PKI since public keys are not transmitted and do not need to be authenticated [13].

Once again, the central issue is that PKE is too costly to implement in a decentralized WSN when network longevity is a priority. Also, the alternative of preloading pairwise symmetric keys for every node pair is inefficient and difficult to manage [13]; however, the idea of linking node IDs to stored keys is a central component of our proposed key management scheme for neighbor discovery.

D. CHAPTER SUMMARY

In this chapter, we provided an overview of the NDP and the security mechanisms implemented by SEND. We then argued against the adequacy of SEND by identifying key issues with its use in decentralized WSNs. A brief overview of related research in key management for secure neighbor discovery concludes the chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SYMMETRIC KEY MANAGEMENT FOR NEIGHBOR DISCOVERY

In this chapter, we discuss our proposed key management scheme for generating and distributing symmetric keys for secure neighbor discovery without a centralized KDC. Our scheme addresses the security and efficiency issues of SEND and the previously described work found in the literature. Our scheme uses the current NDP neighbor discovery messages while implementing symmetric key encryption for confidentiality and energy efficiency. Although we include the Timestamp and Nonce options from the SEND protocol, we replace the CGA option and RSA signature with a protocol for secret link key exchange to protect communications between any pair of nodes. As with identity-based cryptography, the scheme implements node IDs so that each node can associate a link key with the node with which it communicates.

To minimize secret key exposure, each node generates and stores keys only for those nodes within its transmission range rather than preloading a pairwise key for every other node in the network. The scheme also implements a challenge authentication mechanism using a global network key before nodes perform the secret key exchange. The Diffie-Hellman (DH) key exchange algorithm is used for link key generation between nodes due to its relative security and efficiency. Consequently, keys are always locally generated by the nodes rather than wirelessly transmitted between nodes.

A. DIFFIE-HELLMAN KEY EXCHANGE ALGORITHM

Generally, the DH key exchange algorithm allows two entities to generate a single shared secret over an insecure channel without transmitting any secrets [24], [25]. The security of this algorithm relies on the discrete logarithmic problem [24], which is the difficulty of solving for x, given y, g, and p, where

$$y = g^{x} \, mod \, p. \tag{1}$$

The network first establishes two required system parameters, the prime modulus p and the generator g. These very large prime numbers are publicly known and used by all network nodes [24]. When two nodes A and B wish to establish a link, they each choose a random secret number X_A and X_B, respectively [24]. Then, node A computes the public value

$$S_A = g^{X_A} \, mod \, p, \tag{2}$$

while node B computes the public value

$$S_B = g^{X_B} \, mod \, p. \tag{3}$$

Next, nodes A and B exchange S_A and S_B over the channel and use those values to compute the key values K_A and K_B, where

$$K_A = (S_B)^{X_A} \, mod \, p \tag{4}$$

and

$$K_B = (S_A)^{X_B} \, mod \, p. \tag{5}$$

By substituting S_B and S_A in K_A and K_B, respectively, and using the properties of modular exponentiation [25], we find

$$K_A = (g^{X_B} \, mod \, p)^{X_A} \, mod \, p = g^{X_B X_A} \, mod \, p \tag{6}$$

and

$$K_B = (g^{X_A} \, mod \, p)^{X_B} \, mod \, p = g^{X_A X_B} \, mod \, p. \tag{7}$$

Finally, applying the product rule for exponents [25], we conclude the shared key

$$K = K_A = K_B. \tag{8}$$

Therefore, nodes A and B have computed a shared secret key K without transmitting any secret values. Even if an attacker were to intercept any of the transmitted values, the discrete logarithmic problem prevents it from computing the shared key without the secret values X_A and X_B [24].

B. SYMMETRIC KEY MANAGEMENT USING DIFFIE-HELLMAN

As with the NDP, neighbor associations are established and maintained via the neighbor discovery messages; however, with our scheme, these messages are modified to ensure integrity and encrypted to maintain confidentiality. The use of a secret network group key and node IDs provides authentication, while the DH key exchange protocol offers flexible and independent link key management without excessive overheard. Prior to deployment, all nodes are preloaded with the network group key and the node IDs for all nodes in the network.

1. Neighbor Discovery Message Processing

Our scheme modifies the structure of the NDP neighbor solicitation and advertisement messages detailed in [14] to take advantage of identity-based cryptography. The neighbor discovery message follows the Internet Control Message Protocol version 6 (ICMPv6) [15], so it includes an IPv6 header, an ICMPv6 header, the message-specific data, and an options field; however, our scheme simplifies the packet by replacing the CGA option and RSA signature with a node ID. This message format is shown in Figure 2.

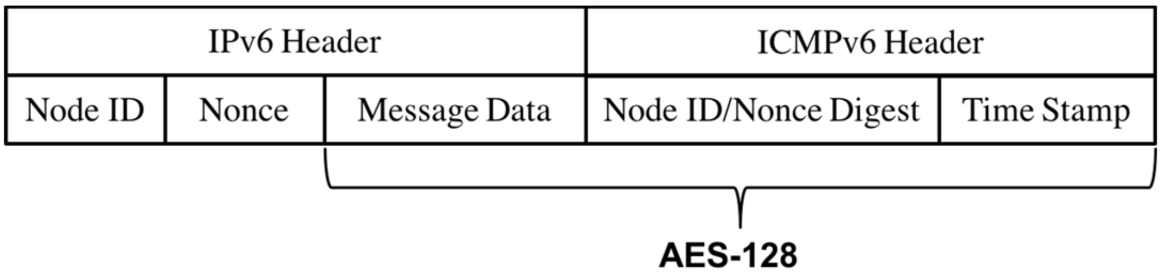

Figure 2. Neighbor Discovery Message Format

Our scheme authenticates a node as a genuine network node when it uses a legitimate node ID and possesses a valid network group key to decrypt messages. All neighbor discovery messages are first processed for authentication when received by a node. After reading the header data per the network routing protocol, the receiving node first checks the node ID for a match in its memory cache. If there is no match, then the sending node is not authenticated, and the message is discarded. When there is a match, the receiving node checks for a link key corresponding to the node ID. If a link key exists between the nodes, then that key is used with AES-128 to decrypt the message data, digest of the node ID and Nonce, and Timestamp. Otherwise, the receiving node defaults to decrypting the encrypted data with the network group key.

After decryption, the receiving node checks for message integrity. First, the node uses the Message Digest 5 (MD5) hash function to compute the hash of the node ID and Nonce and checks the hash against the decrypted digest. Next, it reads the Timestamp and checks whether the elapsed time between the Timestamp and the current time is within a "delta" value that is established by the network administrator. If the message fails either of these checks, then the receiving node discards the message. When the receiving node has successfully authenticated the message and ensured its integrity, the node reads the message data to determine the type of message received.

2. Neighbor Discovery Message Exchange

Our scheme uses the neighbor solicitation and advertisement messages from the NDP but also adds a key request and key exchange message to perform the DH key exchange. Neighbor solicitation messages are addressed as either multicast messages to all network nodes or unicast messages to known nodes. Neighbor advertisement and key exchange messages are always unicast responses to solicitation messages.

a. Multicast Neighbor Solicitation

Each node periodically sends a neighbor solicitation to a multicast address that includes all nodes within the network's address range. The purpose of the multicast solicitation message is to find unknown neighbors within transmission range and initiate key exchange to create neighbor links. The message sequence for multicast neighbor solicitation between node A and an unknown neighbor B is shown in Figure 3.

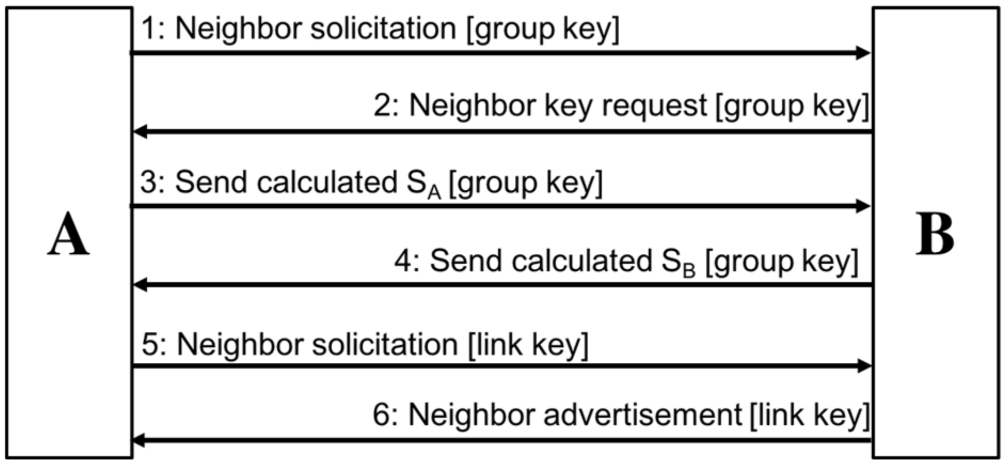
A
B
1: Neighbor solicitation [group key]
2: Neighbor key request [group key]
3: Send calculated S_A [group key]
4: Send calculated S_B [group key]
5: Neighbor solicitation [link key]
6: Neighbor advertisement [link key]

Figure 3. Multicast Neighbor Solicitation to Unknown Neighbor

The multicast neighbor solicitation process begins when node A sends a multicast solicitation to node B, which is always encrypted with the network group key. Node B processes the node ID, discovers it does not have a link key with node A, and decrypts the message with the group key. To establish a link, node B then sends a key request message to node A to initiate the DH key exchange protocol. Upon receipt of the key request, node A calculates S_A and sends it to node B in a key exchange message, while node B calculates S_B and sends it to node A. Nodes A and B then use the received values of S to compute the shared key K and write into memory the key associated with the other's node ID. To complete the link process, node A sends a new *unicast* neighbor solicitation message to node B encrypted with the newly generated link key. Node B processes the node ID, decrypts the message with the link key, and validates the solicitation message. After verifying node A's key in its memory cache, node B responds with a neighbor advertisement message. Finally, Node A processes the advertisement and verifies node B's key in memory.

Since the multicast address includes all network nodes, known neighbors of the sending node also receive the multicast neighbor solicitation messages. The message sequence for known neighbors A and B is shown in Figure 4.

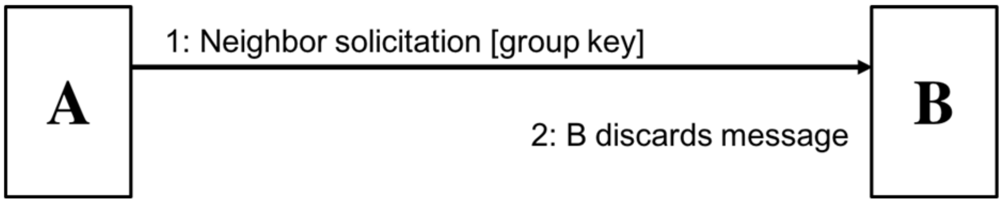

Figure 4. Multicast Neighbor Solicitation to Known Neighbor

Since nodes A and B are known to each other, a pairwise key for this link exists in their respective memory caches. In this case, node A sends the multicast solicitation message, encrypted with the network group key, to node B, a known neighbor. When node B processes the message and identifies node A's ID, it uses their link key to decrypt the message. The decryption will obviously fail, and node B discards the message. Generally, known neighbors always discard multicast neighbor solicitation messages received from each other once the link has been established.

b. Unicast Neighbor Solicitation

Once two nodes have established a link, they rely on periodic unicast neighbor solicitation messages to maintain the link. A unicast neighbor solicitation message can only be sent to known neighbors with known node addresses. The message sequence for unicast neighbor solicitation between known neighbors A and B is shown in Figure 5.

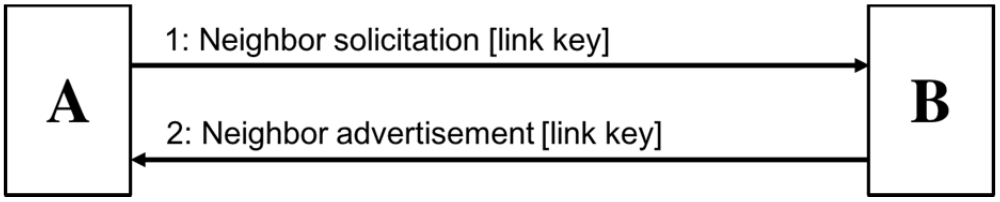
A
1: Neighbor solicitation [link key]
2: Neighbor advertisement [link key]
B

Figure 5. Unicast Neighbor Solicitation to Known Neighbor

Here, node A sends node B a neighbor solicitation message encrypted with their link key. Node B decrypts the message with the same link key, determines that it is a solicitation, and verifies the memory cache entry for node A. Node B then responds with a unicast neighbor advertisement to node A encrypted with the link key. Finally, node A processes the message and verifies its memory cache entry for node B. This completes the unicast neighbor solicitation process.

3. Key Expiration

To protect against secret key exposure, our key management scheme also implements a mechanism that deletes link keys between inactive nodes. This prevents attackers from compromising an inactive node, stealing stored keys, and masquerading as that node in the network. When a node becomes inactive, either by going to sleep or running out of battery power, it stops sending neighbor solicitation messages and responding to its neighbors' solicitations; therefore, its link keys are no longer necessary to maintain the network's neighbor links.

Each node maintains an individual counter associated with every node ID. This counter tracks a node's active neighbors, and it is initialized to zero to represent an inactive node. When node A receives a neighbor advertisement from node B, as depicted in Figure 5, node A sets the counter for node B to a maximum countdown value (e.g., 3). If node A does not receive an advertisement from node B, then it decrements the counter. When a counter reaches zero (e.g., after three failed neighbor solicitation messages), the key attached to the corresponding node ID is deleted from memory.

C. CHAPTER SUMMARY

In this chapter, we described our symmetric key management scheme for secure neighbor discovery. We then explained the mechanics of the DH key exchange algorithm and its application to our scheme. Finally, we discussed our modifications to the NDP message formats and detailed the protocols for processing and exchanging those messages.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. EXPERIMENTAL DESIGN

The simulations for our key management scheme are designed and implemented in MATLAB. The program simulates message transmission and processing, the MD5 hash function, AES-128 encryption and decryption, and the DH key exchange protocol. The simulation is divided into three components: network initialization and node deployment, neighbor discovery and key exchange, and measurement of performance metrics. In our simulations, we measure message efficiency, key distribution, power consumption, and error resistance.

A. NETWORK INITIALIZATION AND NODE DEPLOYMENT

In the first stage of the simulation, we initialize the network and node characteristics given user-input parameters and deploy a random distribution of nodes across the network field.

1. Network Parameters

The simulation receives inputs for several key parameters to provide comprehensive testing of the scheme for a variety of network structures and node characteristics. Users can customize the network field size, total number of nodes, maximum node transmission range, simulation runtime, and network data rate.

2. Node Tracking

The simulation uses a structure array *N* to track information about the network nodes and provide a representation of each node’s memory cache. The *N* array is constantly updated as nodes perform neighbor discovery and key exchange. An example snapshot of the first ten of 100 node entries in *N* after 24 hours of simulated runtime is shown in Table 2.

Table 2. 10-Node Snapshot of Node Tracking Structure Array *N*

Node	State	x	y	nodeID	idStorage	keyStorage	Counter	Power (mWh)
1	1	30	50	‘f0d33b4c…	1×100 cell	1×100 cell	1×100 cell	8.7778
2	1	30	70	‘e1baf3b2…	1×100 cell	1×100 cell	1×100 cell	7.6980
3	1	5	84	‘a75d623…	1×100 cell	1×100 cell	1×100 cell	5.7713
4	1	91	4	‘0834f12e…	1×100 cell	1×100 cell	1×100 cell	3.2976
5	1	90	12	‘701b182…	1×100 cell	1×100 cell	1×100 cell	4.6721
6	1	8	77	‘9b2b8db…	1×100 cell	1×100 cell	1×100 cell	6.3218
7	1	18	42	‘7e6201d4…	1×100 cell	1×100 cell	1×100 cell	6.8511
8	1	97	28	‘c02df246…	1×100 cell	1×100 cell	1×100 cell	5.7589
9	1	20	33	‘84e22dcf…	1×100 cell	1×100 cell	1×100 cell	6.3041
10	1	44	9	‘911465d…	1×100 cell	1×100 cell	1×100 cell	4.3846

The first five columns of *N* track each node’s active or inactive status, coordinate location, and node ID. The *idStorage*, *keyStorage*, and *counter* cell arrays represent each node’s memory cache, which is vital for key management. These cell arrays store node IDs, link keys, and expiration counters for all other nodes in the network. Finally, the *power* value represents the cumulative power in milliwatt-hours consumed by the node at that given time.

3. Node Power Consumption

The simulation uses defined power consumption rates for all simulated node functions. When measuring network energy consumption, we account for the node power consumption due to message transmission and processing, AES-128 encryption, DH key exchange, and the MD5 hash function.

Nodes transmit, receive, and process neighbor discovery messages in the neighbor discovery and key management processes. When a node initiates a neighbor solicitation, it consumes power when transmitting the solicitation message, while the receiving node consumes power receiving and processing the message. The power consumption values for the functions related to messaging are shown in Table 3.

Table 3. Node Power Consumption in Milliwatt-Hours for 127-Byte Message Functions. Adapted from [26] and [27].

Function	Power Consumption
Transmit	8.00×10^{-7} mWh
Receive	5.87×10^{-5} mWh
Process	9.33×10^{-6} mWh

Nodes also consume power when performing AES-128 encryption and decryption on the neighbor discovery messages. When a node sends a message, it must perform AES key setup and encryption, while the receiving node must perform its own key setup and decryption. The power consumption values for the AES-128 functions on 127-byte neighbor discovery messages are shown in Table 4.

Table 4. Node Power Consumption in Milliwatt-Hours for AES-128 Functions on 127-Byte Messages. Adapted from [19] and [26].

Function	Power Consumption
Key Setup	2.18×10^{-6} mWh
Encryption	1.09×10^{-5} mWh
Decryption	2.47×10^{-5} mWh

In the DH key exchange protocol, nodes consume power both in the exchange of *S* values and in the computation of the key *K*. The authors in [19] measured the energy consumption for the DH key exchange algorithm that produces a 1,024-bit key. The resultant conversion to power consumption per node is shown in Table 5.

Table 5. Node Power Consumption in Milliwatt-Hours for 1024-bit DH Key Exchange Algorithm. Adapted from [19].

Function	Key Generation	Key Exchange
DH-1024	0.125 mWh	0.146 mWh

The MD5 hash function is also performed with every message transmission. A transmitting node uses the MD5 function to generate a digest of the node ID and Nonce, while the receiving node uses the MD5 function to compute a hash of the received node

ID and Nonce to check against the digest. In [19], the authors measured the energy consumption of the MD5 algorithm to be 0.59 μJ/B. Since the function processes eight total bytes of data from the node ID and Nonce, the converted power consumption for MD5 is 1.312×10^{-8} mWh.

B. NEIGHBOR DISCOVERY AND KEY MANAGEMENT

After the network is initialized, the main program of the simulation performs the neighbor discovery and key exchange protocols of our scheme. This program runs in looped one-minute iterations for a user-defined *runTime*. In each iteration, the program may perform any of the following tasks: a multicast solicitation, a unicast solicitation, deletion of expired keys, activation of inactive nodes, or measurement of performance metrics.

1. Multicast and Unicast Solicitation

The *multicast* function simulates the entire multicast solicitation message exchange. The *multicast* function first calculates the Euclidean distances between all nodes to determine the nodes that are in transmission range of each other. An encrypted neighbor solicitation message is then assembled using the *neighborSolicit* function and transmitted between all in-range neighbors.

Next, the transmitter's *nodeID* is separated from the message, and the function parses the receiver's *keyStorage* cells for a key matching the *nodeID*. If there is a match, then that link key is used to decrypt. Otherwise, the *GroupKey* is used to decrypt the message. The *packetAuthenticate* function separates the encrypted portion of the message, performs the decryption, validates the Nonce, Timestamp, and message data type, and returns a pass or fail. If the message fails the *packetAuthenticate* function, it is discarded by the receiver. Otherwise, the *DHKey* function is used to perform the DH key exchange between the receiver and transmitter. The resulting key output is then written into the receiver's *keyStorage* cell in *N* and the node's expiration *counter* is set to three, signifying an established link.

Finally, the *neighborSolicit* function sends an advertisement from the receiver back to the original transmitter. The *packetAuthenticate* function is used again to decrypt the packet with the new link key, and after successful authentication, the original transmitter's *keyStorage* and *counter* cells in *N* are updated.

The *unicast* function simulates the entire unicast message exchange and operates very similarly to the *multicast* function. The primary difference is in the method in which the program chooses nodes for message transmission. While the *multicast* function uses transmission range, the *unicast* function parses the *keyStorage* cells in *N* and transmits only to those node IDs that have non-zero attached key values.

2. Key Expiration

The *decrementCounter* function is used to decrement all expiration counters in the node array *N* every 20 minutes. After 60 minutes, or three consecutive failed unicast neighbor solicitations, the *expireKey* function is used to clear the key from the corresponding *keyStorage* cell in the node array *N*.

3. Node Initiation and Activation

When the network is initialized, the simulation only activates a portion of the nodes, leaving the remaining nodes in sleep mode. Random sets of nodes are then activated incrementally to demonstrate the scheme's effectiveness for neighbor discovery and key exchange when new active nodes are introduced to the network. Activation is performed by simply toggling the active status of the selected nodes in *N*.

4. AES-128 and MD5 Hash Function

The *aes* set of functions [28] perform AES key setup, encryption, and decryption in MATLAB. Our simulation uses AES-128 Electronic Codebook Mode (ECB) for encryption and decryption of all transmitted messages. The *DataHash* function [29] performs the MD5 hash function and is used for generating the hash value of the node ID and Nonce in the neighbor discovery messages. This function can receive any type of input and produces a 128-bit hexadecimal string output.

C. MEASUREMENT OF PERFORMANCE METRICS

The performance metrics of message efficiency, key distribution, power consumption, and error resistance are measured throughout the simulation by periodically reading values from *N* and the message counters, which are then compiled and plotted.

1. Message Efficiency

We define message efficiency as the ratio between the number of neighbor solicitation messages sent and the number of neighbor advertisements received by a node. Since there are two types of neighbor solicitation messages in our scheme, unicast and multicast, we measure the efficiency of each solicitation type. A message efficiency closer to one demonstrates that a node is more likely to receive an advertisement in response to a solicitation. An efficiency closer to zero suggests that a node is sending mostly wasted solicitations that are discarded by their recipients. The simulation computes and plots message efficiency every hour using values from the message counters *unicastCount, multicastCount,* and *ackCount*.

2. Key Distribution

Key distribution is the measure of the average number of keys stored per node. This metric is important since we aim to protect against secret key exposure by minimizing the number of keys stored in each node. Every minute, the simulation polls the node array *N* for the total number of active nodes and active keys to determine the average number of active keys stored per node. The simulation then plots the average key distribution as a function of time.

3. Power Consumption

Power consumption is the measure of the average node energy expended over time to perform the functions necessary for message transmission and processing, encryption, and DH key exchange. Using the power consumption rates discussed in the previous chapter, we update the *power* value in *N* after each function is executed in the simulation. We measure and compute the average power consumed per active node once every iteration. The simulation then plots the average node power capacity *remaining* as a function of time.

4. Error Resistance

Our simulation includes a user-defined variable *error* that represents the overall error rate of the model. The *error* value ranges from zero to one and accounts for transmission errors (e.g., dropped packets from collisions, harsh operating environments, etc.), faulty encryption or decryption, or errors in processing. An *error* of zero represents an ideal network environment where all packets are received and processed correctly. According to [30], a packet loss rate of 2.5% is generally acceptable in an IEEE 802.15.4 WSN. The simulation incorporates *error* in the *packetAuthenticate* function, forcing the receiving node to randomly discard a percentage of messages based on the *error* value.

Error resistance is the ability of the key management scheme to withstand network errors and maintain neighbor links. Due to the key expiration mechanism, nodes delete keys after sending multiple unicast neighbor solicitation messages without a neighbor advertisement response. Since the *error* value affects the receipt of both neighbor solicitations and advertisements, nodes are more likely to erroneously delete keys for active links at higher levels of error; therefore, we can use the key distribution plots to evaluate error resistance based on sudden decreases in keys stored per node.

D. CHAPTER SUMMARY

In this chapter, we described the three components of the simulation: the network initialization and node deployment phase, the main program that executes the neighbor discovery and key management scheme, and the measurement and computation of performance metrics.

THIS PAGE INTENTIONALLY LEFT BLANK

V. SIMULATION RESULTS AND ANALYSIS

A series of three simulations was conducted to perform network initialization and node deployment, demonstrate the development of neighbor associations with network growth, and measure the four identified performance metrics. In this chapter, we discuss our simulation results and their significance in improving secure neighbor discovery.

A. NETWORK INITIALIZATION AND NODE DEPLOYMENT

In our first simulation, we set the network size to a 100×100 m^2 field and the simulation time to 24 hours. We used a network of 100 nodes, each with a transmission range of 30 meters and data rate of 250 kbps, which is the maximum data rate for IEEE 802.15.4 [5]. Each node was equipped with a 9-volt battery that provided 4500 mWh of power capacity. All nodes were set to send multicast neighbor solicitation messages every 60 minutes and unicast solicitations every 20 minutes. Network nodes were then deployed in a random distribution within the defined network field, as shown in Figure 6.

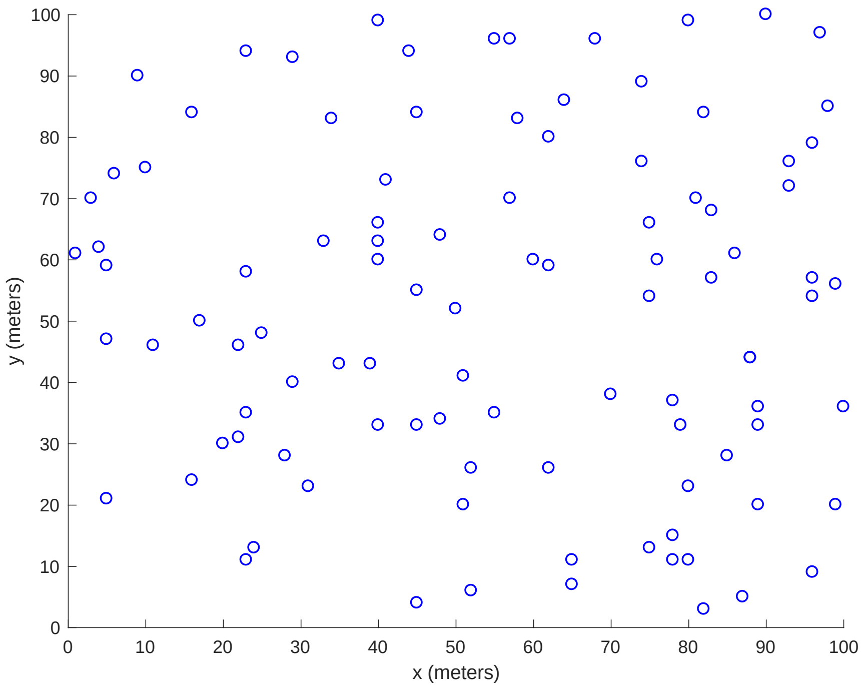

Figure 6. Random Distribution of Nodes in 100×100 m^2 Network Field

B. DEVELOPMENT OF NEIGHBOR ASSOCIATIONS

Although the network was initialized to host 100 nodes, we activated only ten nodes to start, leaving the remaining 90 nodes in sleep mode. Every five hours, we conducted a node activation cycle where an additional 25 randomly selected nodes were activated. The simulation shows that our scheme successfully generated links for all newly discovered neighbors after each activation cycle. The development of the network's neighbor associations is shown in Figures 7 through 10, where the lines represent pairwise keys shared between the connected nodes. This series of plots shows a significant improvement in the network infrastructure as new nodes are activated and links are created to improve routing efficiency and reliability. These links were generated without a centralized BS, which increases flexibility and decreases overall energy costs for messaging.

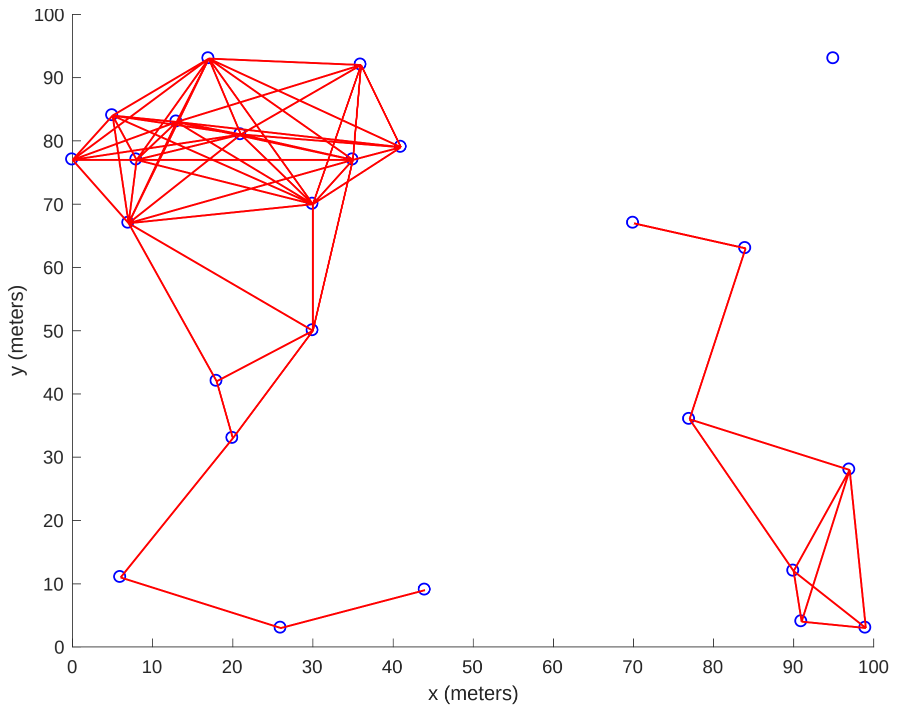

Figure 7. Neighbor Associations for 25 Active Nodes at Time = 6 Hours

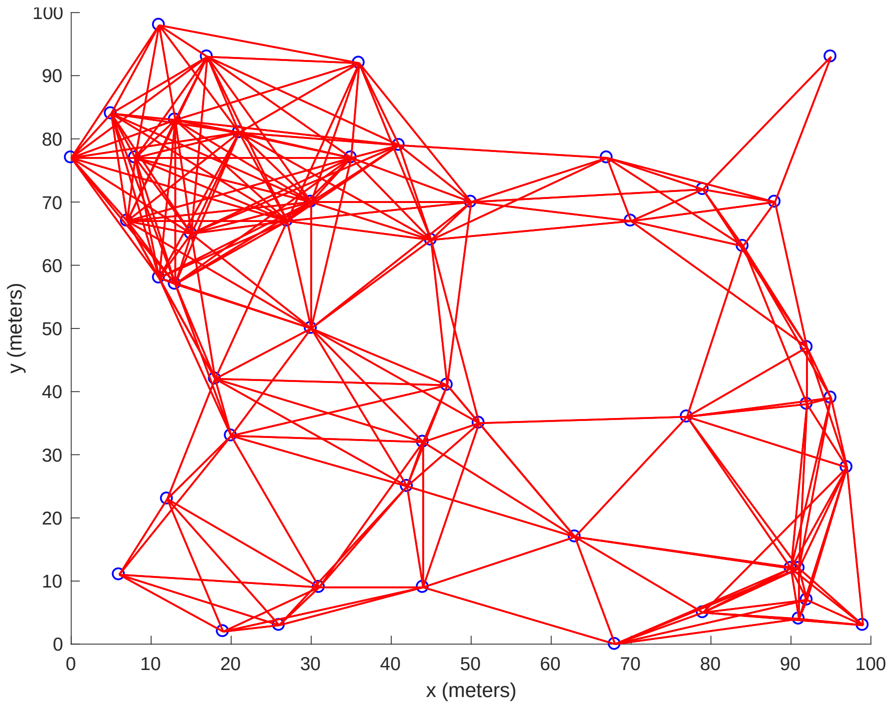

Figure 8. Neighbor Associations for 50 Active Nodes at Time = 12 Hours

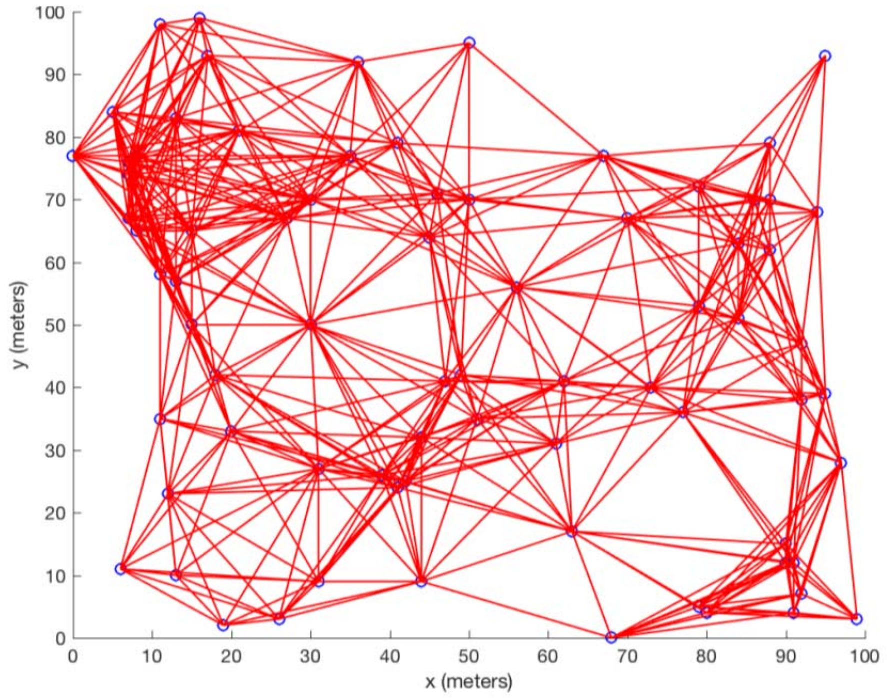

Figure 9. Neighbor Associations for 75 Active Nodes at Time = 18 Hours

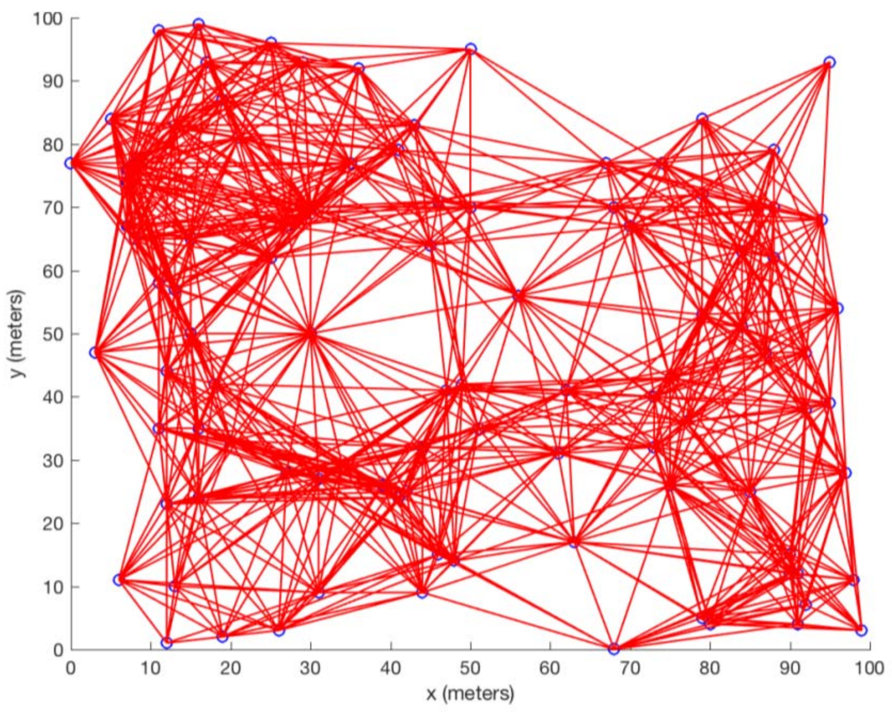

Figure 10. Neighbor Associations for 100 Active Nodes at Time = 24 Hours

C. MESSAGE EFFICIENCY

Message efficiency was measured based on the number of multicast solicitations, unicast solicitations, and advertisements sent by all nodes in the 24-hour simulation period. The plot of cumulative message traffic is shown in Figure 11. From Figure 11, we find a strong correlation between the number of unicast solicitation messages and advertisements. This correlation suggests that the transmission of neighbor advertisements is more dependent on unicast solicitations than multicast solicitations. This correlation is even more clear in Figure 12, where we show the message efficiency for both multicast and unicast solicitations in an ideal, error-free simulation.

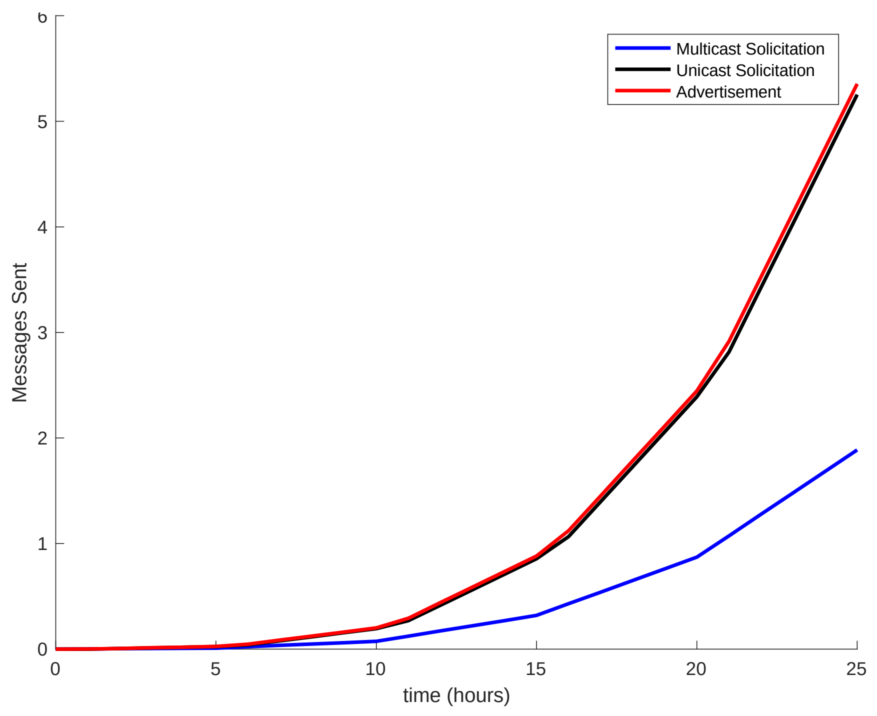

Figure 11. Cumulative Neighbor Discovery Message Traffic After 24 Hours

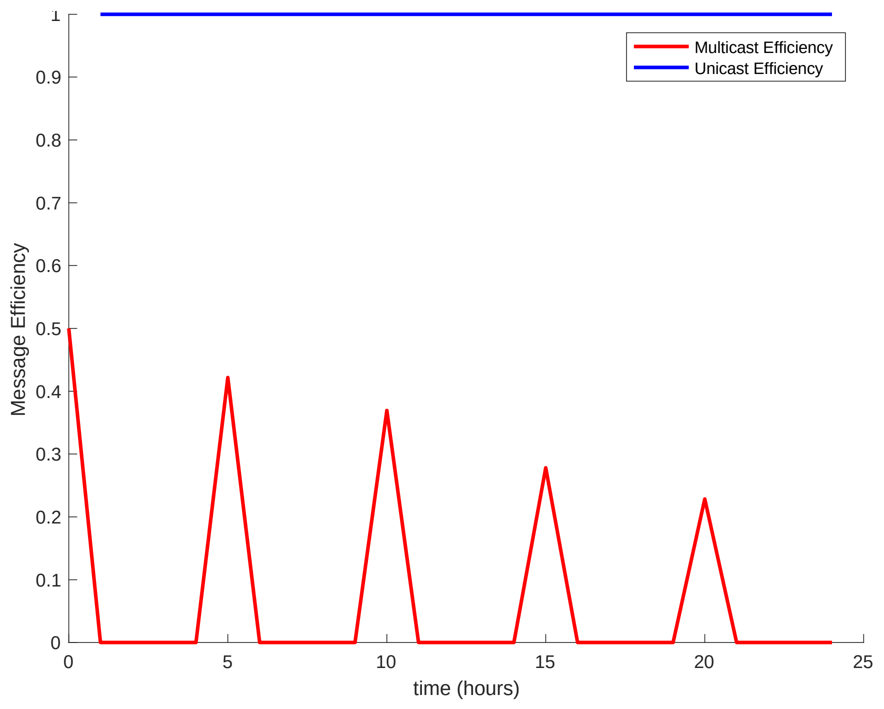

Figure 12. Multicast and Unicast Message Efficiency Over 24-Hour Simulation

From Figure 12, we see that multicast solicitations are only used to discover new neighbors and initiate key exchange at each node activation cycle. As expected from our scheme, nodes discard all multicast solicitations once their links have been established, resulting in the multicast efficiency of zero between node activation cycles. The message efficiency for unicast solicitations shows that our scheme is heavily reliant on unicast solicitations for maintaining neighbor links. As static networks become increasingly stable, we can substantially reduce the frequency of multicast messages to save node energy and increase network longevity.

D. KEY DISTRIBUTION

The key distribution of the network stayed very consistent throughout the simulation. On average for a 100×100 m^2 network, each node only needed to store pairwise keys for about 20% of the active nodes in the network even after multiple node activation cycles. The average number of keys stored per node for each active network size is shown in Table 6.

Table 6. Average Number of Keys Stored Per Node Over 24-Hour Simulation

Total Active Nodes	Average Keys Per Node	Key Percentage
10	2	20.0%
25	5.12	20.5%
50	9.80	19.6%
75	14.72	19.6%
100	20.32	20.3%

This key distribution is a marked improvement in efficiency over the SEND protocol, which uses PKE in its neighbor discovery messaging. In PKE, nodes store public keys for all other nodes in the network; however, due to the limited transmission range of the nodes, only a small percentage of those keys are useful. Since our scheme only generates keys between nodes in range of each other, we infer that only 20% of the network is ever in range of any given node; therefore, 80% of the public keys stored in each node go unused if the SEND protocol is used.

The active node count and average number of keys stored per node are also graphically depicted in Figure 13. The drop-off in average number of keys stored per node every five hours indicates the short time immediately following a node activation cycle when new keys had not yet been generated.

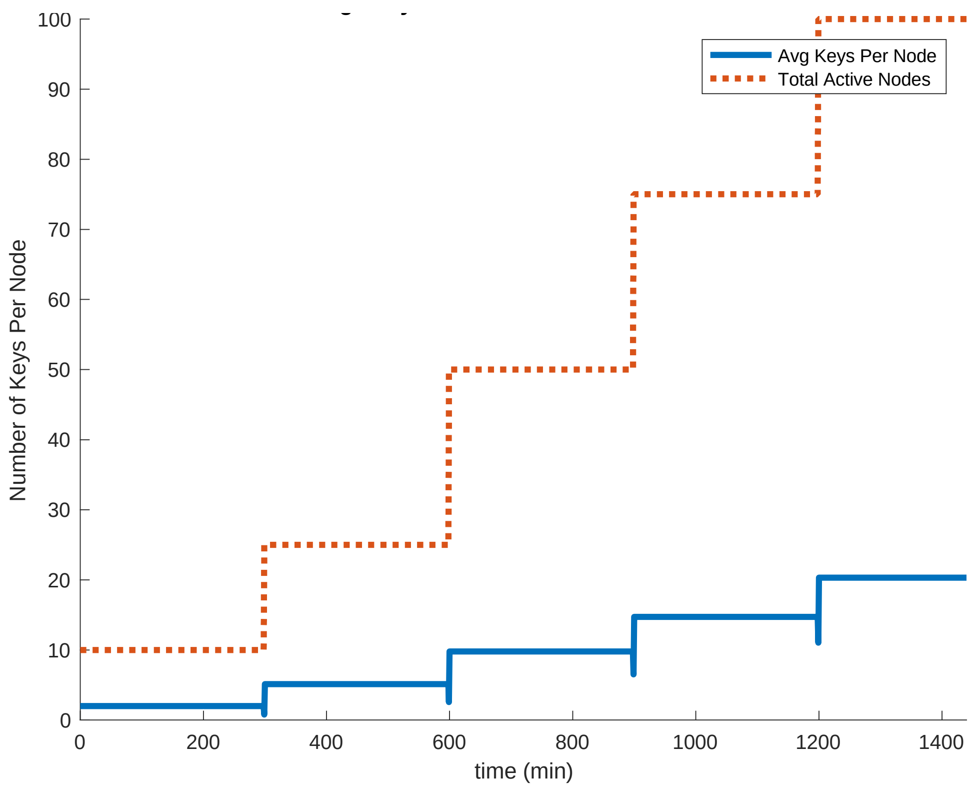

Figure 13. Average Number of Keys Stored per Node Over 24-Hour Simulation

These results confirm that our scheme protects against the threat of secret key exposure by limiting the number of keys stored by an active node. If any single node was to be compromised, the resulting security breach would be contained to no more than 20% of the network, regardless of the network size. Additionally, by reducing the number of keys stored in memory, we conserve the limited memory resources inherent to IEEE 802.15.4 enabled devices.

E. POWER CONSUMPTION

Our second set of simulations measured the average node power consumption for three network sizes of 25, 50, and 100 nodes. The simulation was run until all nodes had fully expended their battery power. Since all nodes were equipped with 9 V batteries, each node started with 4500 mWh of power capacity. The average power consumption per node for all three networks is depicted in Figure 14.

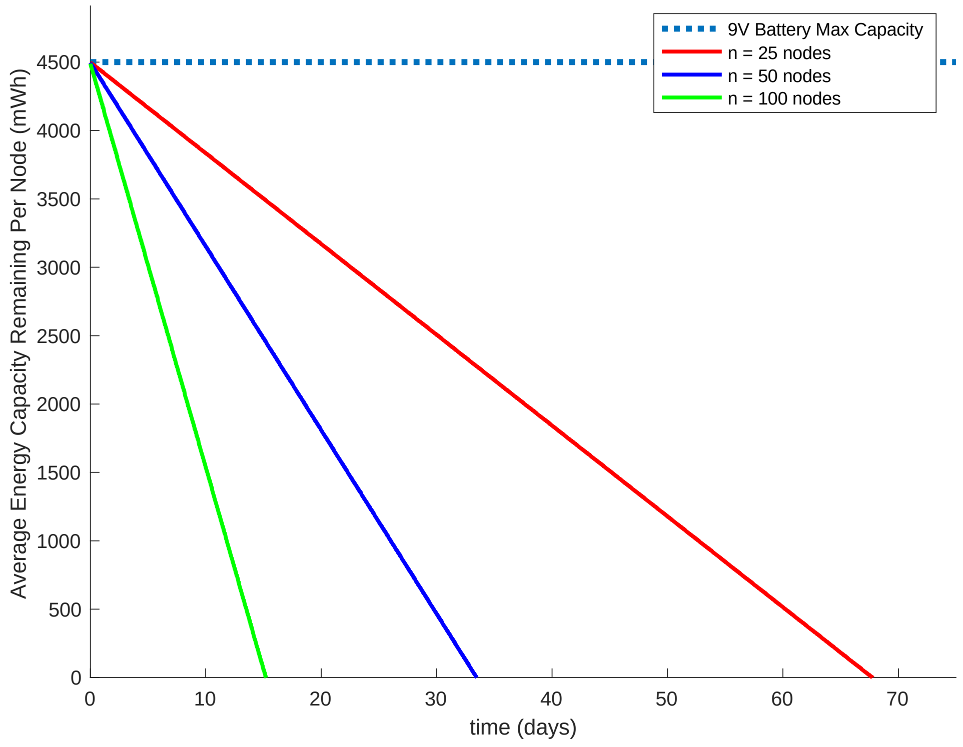

Figure 14. Average Power Consumption for Networks of 25, 50, and 100 Nodes

From Figure 14, we see that smaller network sizes have greater longevity since there are fewer neighbor associations to maintain. Generally, doubling the network size halves the node power consumption. The 100-node network required the most energy and lasted only about 15 days. Meanwhile, the 50-node network lasted about 33 days, and the 25-node network lasted nearly 70 days.

According to [2], the individual sensors of the TRSS are expected to operate continuously for up to 30 days; therefore, the 50-node network can provide the same 30-day period of operation with a much more robust and redundant coverage of one node every four meters. Also, as previously discussed, we can decrease power consumption by reducing the frequency of neighbor discovery messages, or we can simply reduce the number of active nodes in the network. In the case of the 50-node network, we can set half the network to sleep mode and double the network longevity while still maintaining effective coverage of one node every eight meters.

F. ERROR RESISTANCE

In our final set of simulations, we measured the effects of error on our scheme. We initialized a 100×100 m^2 network of 50 nodes with a run time of 12 hours. We then performed the simulation for *error* values of 0, 0.05, 0.10, and 0.20 and evaluated the error resistance from the key distribution plots for each *error* value, which are shown in Figures 15 through 18. This series of plots shows a gradual degradation in the ability of the network to maintain neighbor associations as the error rate increased.

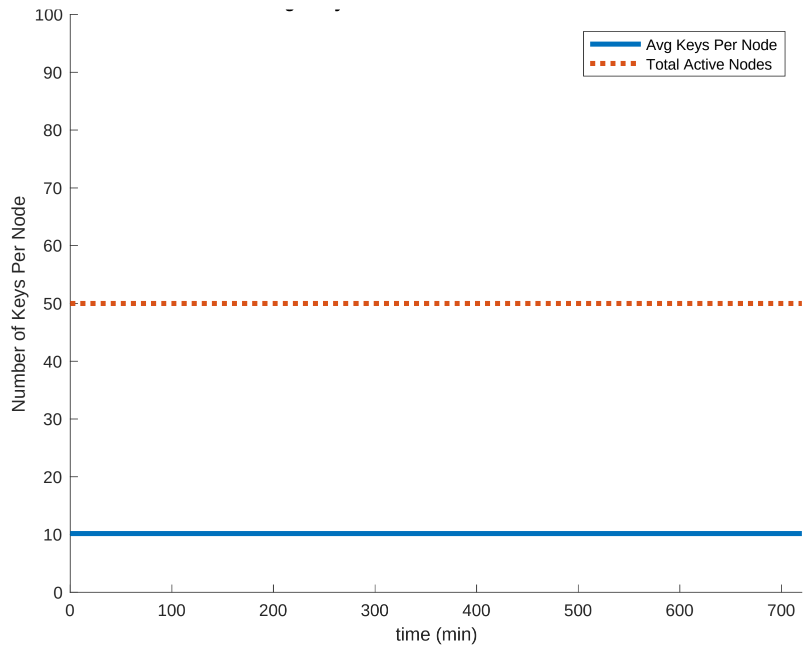

Figure 15. No Degradation of Key Distribution for *Error* = 0%

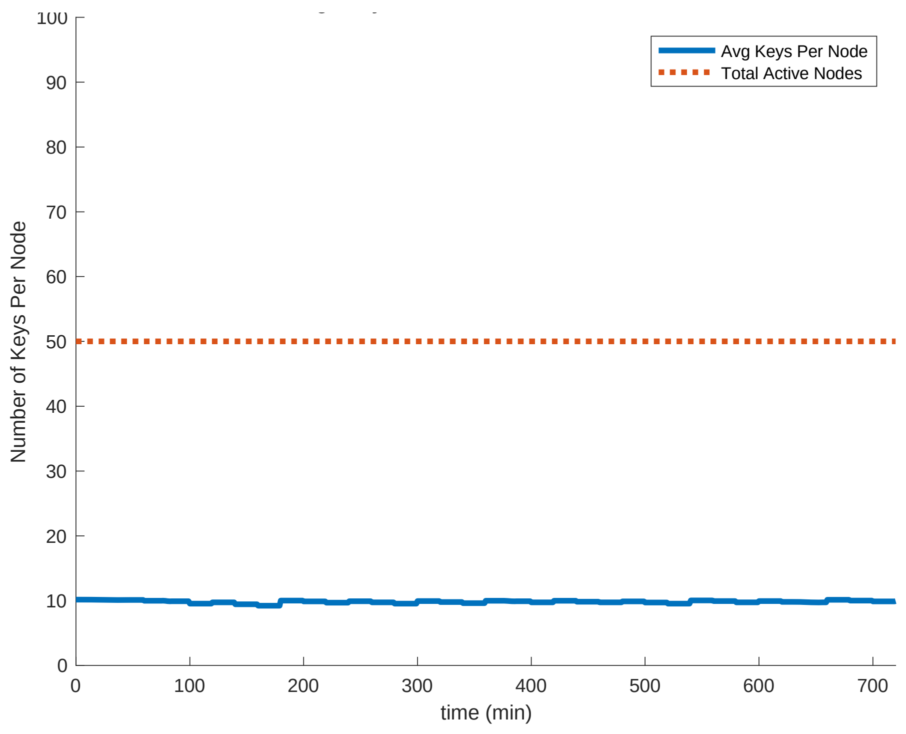

Figure 16. Minimal Degradation of Key Distribution for *Error* = 5%

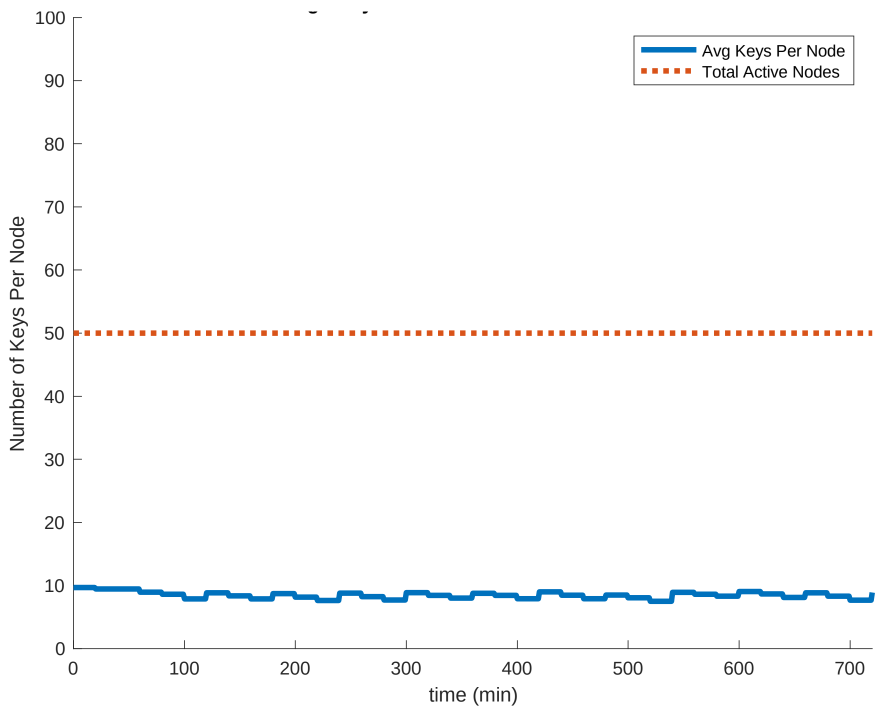

Figure 17. Moderate Degradation of Key Distribution for *Error* = 10%

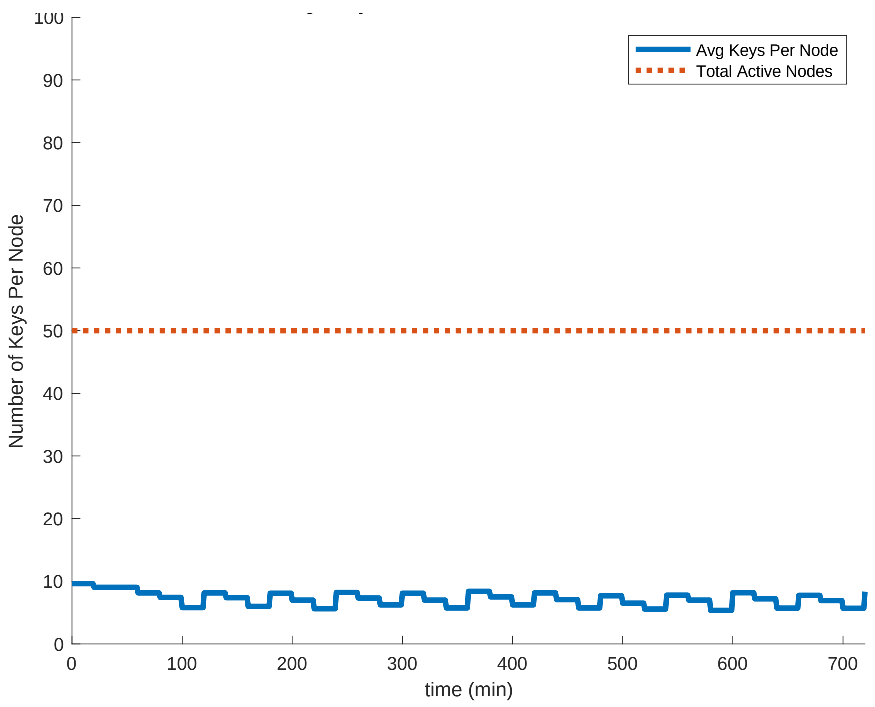

Figure 18. Significant Degradation of Key Distribution for *Error* = 20%

In the ideal case shown in Figure 15 where no messages were discarded due to error, no keys were erroneously deleted from the nodes. From Figure 16, at 5% *error*, which is twice the acceptable packet loss rate of 2.5%, the effects on key distribution were minimal. From Figures 17 and 18, we see that at higher error levels of 10% and 20%, the network performance was significantly degraded as nodes lost many of their neighbor links. The simulation shows that our scheme is able to adequately maintain network links given a reasonable and expected level of packet loss inherent to wireless communications.

We also note that at higher expected error levels, increasing the frequency of solicitation messages improves performance and prevents active keys from expiring. Conversely, at lower expected error levels, fewer solicitation messages can be sent while maintaining the same performance and improving network energy efficiency.

G. CHAPTER SUMMARY

In this chapter, we reviewed the results of our simulations and provided analysis of our key performance metrics, to include message efficiency, key distribution, power consumption, and error resistance.

VI. CONCLUSIONS AND FUTURE WORK

A. SUMMARY AND CONCLUSIONS

The use of WSN will only increase as the focus of our C2 infrastructure shifts toward a highly secure, low-signature solution that can provide commanders with accurate and timely information in the most austere operating environments. Current sensor systems do not provide adequate security against a growing network threat. Our research was motivated by the need to provide security to one of the most fundamental functions for maintaining a WSN while adhering to the operating constraints of IEEE 802.15.4 enabled devices.

Our symmetric key management scheme for secure neighbor discovery was designed to address the security issues of the NDP and the efficiency issues of SEND and other related work in secure neighbor discovery in WSN. In this scheme, we modified the NDP messages and SEND protocol to use identity-based symmetric key encryption rather than PKE. More importantly, we implemented the DH key exchange algorithm to perform secret key exchange between end nodes without the use of a central BS. These adaptations significantly improved the energy efficiency of the neighbor discovery process and eliminated the need for a centralized infrastructure for neighbor discovery.

Our scheme was simulated in MATLAB to demonstrate that it could effectively perform secure neighbor discovery. The simulation showed that the scheme protected against secret key exposure by effectively minimizing the number of keys stored in each node. We also showed that our scheme supported continuous operation of low-powered devices to meet USMC operational requirements for a sensor system. Finally, our simulation indicated that the scheme was tolerant to transmission and processing errors.

B. CONTRIBUTIONS OF THIS THESIS

Our objective was to develop a key management protocol for secure neighbor discovery in WSN that could function on resource-constrained IEEE 802.15.4 enabled devices. In this thesis research, we have contributed the following to the study of key management in WSN:

- Development of a symmetric link key management scheme for WSN that generates and maintains link keys without a centralized node.
- Simulation of the key management scheme to validate its effectiveness in finding one-hop neighbors and generating and maintaining link keys while staying within the operating constraints.
- Measurement of the performance of the key management scheme for message efficiency, key distribution, power consumption, and error resistance.

C. FUTURE WORK

We demonstrated that symmetric keys can be generated and managed solely by end node devices in a static network; however, there are potential areas for further research that can extend the utility of the scheme to mobile networks and validate it against common forms of attack.

1. Mobile Sensor Networks

The simulations showed that our key management scheme was effective in static WSN environments; however, the USMC will likely leverage WSN applications with mobile devices that require the same level of security for neighbor discovery. As a result, our scheme should be refined and optimized for mobile WSN. Considerations include improving the efficiency of the messaging scheme for mobile nodes and ensuring the prompt generation and expiration of keys as nodes move in and out of range of each other.

2. Validation against Common Forms of Attack

Our research served as a proof-of-concept for our key management scheme as applied to neighbor discovery; however, the scheme has not been subjected to various forms of attack such as man-in-the-middle, identity spoofing, or compromised-key attack. Simulations should be designed and implemented to validate the security of the scheme against these attacks and assess its ability to provide higher levels of confidentiality and integrity to the neighbor discovery process.

APPENDIX. MATLAB SIMULATION CODE

```
%%  SIMULATION.M

%   Symmetric key generation for secure mutual authentication of
%   decentralized nodes in a 802.15.4 wireless sensor network.

%   Captain Kelvin T. Chew
%   Student, M.S. Electrical Engineering
%   Naval Postgraduate School

%   STAGE 1: Network Initialization and Node Deployment
%   STAGE 2: Main Program in Time Domain
%   STAGE 3: Computation of Performance Metrics

%%  STAGE 1: NETWORK INITIALIZATION AND NODE DEPLOYMENT

clear
clc

%  Declare global variables

global TransRange           % maximum transmission range
global N                    % node structure
global n                    % active nodes in the network
global runTime
global totalNodes
global DataRate
global error
global errorCount
global GroupKey
global adMessage
global ackMessage
global energy
global adCount
global ackCount
global discardCount
global broadcastCount
global multicastCount
global messageCount
global errorCounter
global keyCount
global nodeCount
global totalKeys
global avgNodeKeys
global totalEnergy
global avgNodeEnergy
global plotCount

%  Set global network parameters

n = 100;%input('Network Size (nodes)?  ');  % number of start
active nodes
totalNodes = 100;    % maximum active nodes
```

```
runTime = 12*60;%input('Run Time?  ');  % Simulation Run Time
(seconds)

field = 100;%input('Field Size (n x n)?  ');  % size of the
sensor field

TransRange = 30;%input('Max Transmission Range?  ');  % max
transmission range

DataRate = 250000;  % max 250 kbps data rate for 805.15.4

error = 0.05;      % determines overall model error rate (0-1)
errorCount = 0;

keyCount = 0;       % tracks active symmetric keys

GroupKey = DataHash(rand()); % shared pre-loaded group key,
generated from
                             % MD5 hash of randomly generated
number

adMessage = sprintf('%02x','advertise');        % HEX string of
ad text
ackMessage = sprintf('%02x','acknowleg');       % HEX string of
ack text

adCount = 0;        % tracks number of multicast advertisements
sent
ackCount = 0;       % tracks number of acknowledgements sent
discardCount = 0;   % tracks number of packets discarded
multicastCount = 0;   % tracks number of unicast advertisements
sent
broadcastCount = 0;
messageCount = cell(4,runTime/60);
errorCounter = cell(1,runTime/60);
nodeCount = zeros(1,runTime+1);       % tracks active nodes at
given time
totalKeys = zeros(1,runTime+1);       % tracks total keys in
network
avgNodeKeys = zeros(1,runTime+1);     % tracks avg keys per node
totalEnergy = zeros(1,runTime+1);     % tracks total energy in
network
avgNodeEnergy = zeros(1,runTime+1);   % tracks avg energy consumed
per node

energy.transmit = 8*10^-7;              % all values in mWh
energy.receive = 5.867 * 10^-7;
energy.process = 9.33*10^-6;
energy.aes_key = 2.18*10^-6;            % 8 128-bit blocks per
packet for
energy.aes_encrypt = 1.09 * 10^-5;      % AES functions: key
scheduling,
energy.aes_decrypt = 2.47 * 10^-5;      % encryption, and
decryption
energy.MD5 = 1.64 * 10^-9 * 8;
energy.DH_key_generation = .249 / 2;    %  DH for 1024-bit key
energy.DH_key_exchange = .291 / 2;      %  Value halved per node
energy.node_energy = 4500;                      % capacity of 9V
battery

%  Create data structure N for all nodes that contains their
parameters

for index = 1:totalNodes
    N(index).power = 0;                   % initialize all nodes
powered off
    for count = 1:n                       % initialize powered on
nodes
        N(count).power = 1;
    end
    N(index).x = round(rand(1)*field);    % x-coordinate of node in
field
    N(index).y = round(rand(1)*field);    % y-coordinate of node in
field
    N(index).nodeID = createNodeID(index);    % preloaded Node ID
    N(index).idStorage = cell(1,totalNodes);     % stores all
neighbor IDs
    for count = 1:totalNodes                  % initialize ID
storage to '0'
        N(index).idStorage{count} = 0;
    end
    N(index).keyStorage = cell(1,totalNodes);    % stores
symmetric keys
    N(index).checkFlag = cell(1,totalNodes);  % check flag for
node timeout
    for count = 1:totalNodes                % initialize check
flags to '0'
        N(index).checkFlag{count} = 0;
    end
    N(index).energy = 0;                  % node energy consumed
end

%%  STAGE 2: MAIN PROGRAM IN TIME DOMAIN

broadCount = 0;
multiCount = 1;
decrementCount = 1;
expireCount = 1;
addNodeCount = 1;
removeNodeCount = 1;
plotCount = 0;
messageCounter = 0;
addNodes = 15;
removeNodes = 10;

for time = 0:runTime          % time measured in minutes

    % Neighbor discovery phase: broadcast every 5 seconds
starting at t=0

    while time == 60 * broadCount
        multicast
        broadCount = broadCount + 1;
    end
    % Neighbor advertisement phase: multicast every 1 second
starting t=2

    while time == 20 * multiCount
        unicast
        multiCount = multiCount + 1;
    end

    % Periodic decrement of all node check flags (every 2
seconds)

    while time == 20 * decrementCount
        decrementCheck
        decrementCount = decrementCount + 1;
    end

    % Deletion of expired nodes with 6 second timeout (every 2
seconds)

    while time == 20 * expireCount
        expireNode
        expireCount = expireCount + 1;
    end

    % Turn on 15 new nodes (at 10 and 20 seconds)

    while time == 5 * 60 * addNodeCount - 1
        %if time > 20
        %    break
        %end
        for index = (n + 1):(n + addNodes)
            N(index).power = 1;                    % power on node
            N(index).x = round(rand(1)*field);   % x-coordinate of
node in field
            N(index).y = round(rand(1)*field);   % y-coordinate of
node in field
            N(index).nodeID = createNodeID(index);    % preloaded
Node ID
            N(index).idStorage = cell(1,totalNodes);  % stores
neighbor ID
            for count = 1:totalNodes         % initialize ID
storage to '0'
                N(index).idStorage{count} = 0;
            end
            N(index).keyStorage = cell(1,totalNodes);    % stores
symmetric neighbor keys
            N(index).checkFlag = cell(1,totalNodes);     % check
flag for ID time out
            for count = 1:totalNodes        % initialize ID
storage to '0'
                N(index).checkFlag{count} = 0;
            end
            N(index).energy = 0;                   % node energy
remaining
        end
        n = n + addNodes;
        addNodeCount = addNodeCount + 1;
        addNodes = 25;
    end

    % Plot network after adding nodes to include all active nodes
and 1-hop
    % neighbor associations.  Neighbor associations are based on
the
    % existence of a shared key between nodes.

    while time == 6 * 60 * plotCount        % new plot every 6
hours
        figure(plotCount+1)
        hold on

        axis([0 field 0 field])
        xlabel('x (meters)')
        ylabel('y (meters)')
        title(['Wireless Sensor Network at time = '
num2str(time/60) ' Hours'])

        for i = 1:totalNodes
            if N(i).power == 1
                plot(N(i).x,N(i).y,'bo');
            end
        end

        for node = 1:totalNodes
            for i = 1:totalNodes
                if N(node).checkFlag{i} > 0
                    line([N(node).x,N(i).x],[N(node).y,N(i).y],'c
olor','r','LineWidth',1);
                end
            end
        end

        hold off
        plotCount = plotCount + 1;
    end

    % Calculate total and average shared keys per node and the
total and
    % average energy consumption per node

    for node = 1:totalNodes

        totalEnergy(time+1) = totalEnergy(time+1) +
N(node).energy;

        for i = 1:totalNodes
            if isempty(N(node).idStorage{i})
            else
                if N(node).idStorage{i} == 0
                else
                    keyCount = keyCount + 1;
                end
            end
        end
    end

    %  Tabulate total messages sent everyone hour

    while time == 60 * messageCounter
        messageCount{1,messageCounter+1} = messageCounter + 1;
        messageCount{2,messageCounter+1} = broadcastCount;
        messageCount{3,messageCounter+1} = multicastCount;
        messageCount{4,messageCounter+1} = ackCount;
        errorCounter{messageCounter+1} = errorCount;
        messageCounter = messageCounter + 1;
    end


    avgNodeEnergy(time+1) = energy.node_energy - 
(totalEnergy(time+1) / n);

    totalKeys(time+1) = keyCount;
    nodeCount(time+1) = n;
    avgNodeKeys(time+1) = keyCount / nodeCount(time+1);
    keyCount = 0;

end

%%  STAGE 3: PLOT AND PRINT RESULTS OF PERFORMANCE METRICS

    %  Plot the average keys stored per node over the entire run 
time

    t = 0:runTime;
    figure(plotCount + 1)
    hold on

    axis([0 runTime 0 100])
    xlabel('time (min)')
    ylabel('Number of Keys Per Node')
    title('Average Keys Stored Per Node Over 12 Hours')

    plot(t,avgNodeKeys,'LineWidth',3)
    plot(t,nodeCount,'LineStyle',':','LineWidth',3)

    legend('Avg Keys Per Node','Total Active Nodes')

    hold off
    plotCount = plotCount + 1;

*****************************************************************

function [] = multicast()

global N
global n
global GroupKey
global broadcastCount
global ackCount
global discardCount
global energy

for TX = 1:n                            % outer for loop for
transmitting nodes
    for RX = 1:n                        % inner for loop for
receiving nodes

        if (inRange(N(TX).x,N(TX).y,N(RX).x,N(RX).y)) && (TX ~=
RX)

            % transmitted encrypted packet with 72-byte payload
            TX_packet = neighborAd(N(TX).nodeID,GroupKey);

            N(TX).energy = N(TX).energy + energy.transmit +
energy.MD5 + energy.aes_encrypt + energy.aes_key;

            % RX node will read the Node ID from packet and use
the
            % appropriate key.  If the TX Node ID is found in
memory, then
            % it will use the key associated with that
ID.  Otherwise, it
            % will use the group key to decrypt.

            RX_key = findKey(RX,TX_packet);
            broadcastCount = broadcastCount + 1;
            N(RX).energy = N(RX).energy + energy.receive +
energy.process;

            % RX node decrypts transmitted packet and
authenticates
            % payload.  If the payload is authentic, then RX will
read the
            % message type.

            N(RX).energy = N(RX).energy + energy.MD5 +
energy.aes_key + energy.aes_decrypt;

            if packetAuthenticate(TX_packet,RX_key) == 1

                % RX reads advertisement.  If TX node ID is not
in RX
                % memory, then RX will send key request to
initiate
                % DH key exchange.  Once the shared secret key
between RX
                % and TX has been generated, the node IDs and key
are
                % written into each node's memory.

                if checkNodeID(RX,TX,TX_packet,1) ~= 0
                    disp('RX Key already exists')
                end

                if checkNodeID(RX,TX,TX_packet,2) ~= 0
                    disp('TX Key already exists')
                end

                if checkNodeID(RX,TX,TX_packet,1) == 0   % no key
in RX

                    keyExchange = DHKey(randi(15),randi(15));
                    N(RX).energy = N(RX).energy +
energy.DH_key_generation + energy.DH_key_exchange;
                    N(RX).energy = N(RX).energy + 2 *
energy.aes_encrypt + 2 * energy.aes_decrypt + 4 * energy.aes_key;
                    N(RX).energy = N(RX).energy + 2 *
energy.transmit + 2 * energy.receive + 2 * energy.process;
                    N(TX).energy = N(TX).energy +
energy.DH_key_generation + energy.DH_key_exchange;
                    N(TX).energy = N(TX).energy + 2 *
energy.aes_encrypt + 2 * energy.aes_decrypt + 4 * energy.aes_key;
                    N(TX).energy = N(TX).energy + 2 *
energy.transmit + 2 * energy.receive + 2 * energy.process;

                    N(RX).idStorage{TX} = N(TX).nodeID;
                    N(RX).keyStorage{TX} = keyExchange;

                end

                if checkNodeID(RX,TX,TX_packet,2) == 0 % no key
in TX

                    N(TX).idStorage{RX} = N(RX).nodeID;
                    N(TX).keyStorage{RX} = keyExchange;

                end

                %  RX sends acknowledgement back to TX

                RX_packet = neighborAck(RX,TX);
                N(RX).energy = N(RX).energy + energy.transmit +
energy.MD5 + energy.aes_key + energy.aes_encrypt;

                %  TX reads acknowledgement

                N(TX).energy = N(TX).energy + energy.receive +
energy.process + energy.MD5 + energy.aes_key +
energy.aes_decrypt;

                if
packetAuthenticate(RX_packet,findKey(TX,RX_packet)) == 1
                    N(TX).checkFlag{RX} = 3;
                    N(RX).checkFlag{TX} = 3;
                    ackCount = ackCount + 1;
                end

            else
                discardCount = discardCount + 1;  % RX discards
packet
            end
        end
    end
```

```
end

end

*****************************************************************

function [] = unicast()

global N
global n
global ackCount
global discardCount
global multicastCount
global energy
global totalNodes

for TX = 1:totalNodes    %  The outer loop will cycle through
each node's multicast

    for RX = 1:totalNodes       % Inner loop simulates single
multicast

        if (isempty(N(TX).keyStorage{RX})) == 0  % ad sent for
every key

            TX_packet =
neighborAd(N(TX).nodeID,N(TX).keyStorage{RX});

            multicastCount = multicastCount + 1;

            N(TX).energy = N(TX).energy + energy.transmit +
energy.MD5 + energy.aes_encrypt + energy.aes_key;

            RX_key = findKey(RX,TX_packet); % RX finds shared key
in mem

            N(RX).energy = N(RX).energy + energy.receive +
energy.process;

            N(RX).energy = N(RX).energy + energy.MD5 +
energy.aes_key + energy.aes_decrypt;

            if packetAuthenticate(TX_packet,RX_key) == 1

                RX_packet = neighborAck(RX,TX);
                N(RX).energy = N(RX).energy + energy.transmit +
energy.MD5 + energy.aes_key + energy.aes_encrypt;

                N(TX).energy = N(TX).energy + energy.receive +
energy.process + energy.MD5 + energy.aes_key +
energy.aes_decrypt;
                if
packetAuthenticate(RX_packet,findKey(TX,RX_packet)) == 1
                    N(TX).checkFlag{RX} = 3;
                    ackCount = ackCount + 1;
                end
            else
                disp('Error')
            end
        end
    end

end

end

*****************************************************************

function [authentic] = packetAuthenticate(packet_data,key)

%%  PACKET AUTHENTICATION FUNCTION

%   This function receives as input a packet of data and a
key.  It then
%   separates the packet into the unencrypted nonce and the
encrypted
%   payload.  The payload is then decrypted with AES-128 using
the key.
%   The unencrypted nonce is compared with the decrypted nonce to
%   authenticate the message.  Error from all sources is
incorporated into
%   this function.

global error
global errorCount

nonce = packet_data(33:40);      % separate nonce from packet data
payload = packet_data(41:72);     % separate payload from packet
data

plain_text = aes_decrypt(payload,key);     % decrypt payload

if plain_text(19:26) == lower(nonce)   % check to see that nonce
sent in
    if rand < (1 - error)
        authentic = 1;              % the clear matches nonce in
encrypted
    else authentic = 0;
        errorCount = errorCount + 1;
    end
else
    authentic = 0;% payload, considering the padding of zeros
end

end

*****************************************************************

function [key] = DHKey(a_secret,b_secret)
%% Diffie-Hellman Key Exchange
%  This function will compute the shared secret key between two
nodes A and
```

```
%  B using the Diffie-Hellman key exchange algorithm.  Due to the
%  the limitations of MATLAB's modulus function, the secret
number of each
%  node is limited to a value from 0 to 15.  The input of the
function
%  will be in HEX as "0_".  The resulting shared secret will
range from 0
%  to 15, and a MD5 hash value is then computed to output a
shared 128-bit
%  key for AES encryption between neighbors.

%% Main Function

p = 13;     % These values for p & g are chosen based on the
MATLAB limits.
g = 2;

a = a_secret;
b = b_secret;

A = mod(g^a,p);
B = mod(g^b,p);

A_Key = mod(B^a,p);
B_Key = mod(A^b,p);

if A_Key == B_Key
    shared_key = DataHash(A_Key);
else
    disp('Error')
end
%% MD5 Hash of Shared Secret to Produce 128-bit AES Key

key = DataHash(shared_key);

*****************************************************************

function [key] = findKey(RX_Number,packet_data)

%%  FIND KEY FUNCTION

%   This function enables a RX node to read the node ID from a
packet and
%   parse its memory for a shared symmetric key.  If a shared key
exists in
%   memory, then the node will use that key for encryption and
decryption.
%   Otherwise, it will use the group key.  The function returns
the appropriate key.

global N
global n
global GroupKey
global totalNodes

nodeID = packet_data(1:32);
```

```
key_found = 0;
key = GroupKey;

for i = 1:totalNodes
    if key_found == 0
        if nodeID == N(RX_Number).idStorage{i}
            key = N(RX_Number).keyStorage{i};
            key_found = 1;
        end

    end
end

*****************************************************************

function [packet] = neighborAck(RX,TX)

%%  NEIGHBOR ACKNOWLEDGEMENT FUNCTION

%   This function encrypts a 127-octet packet using AES-128 with
the paired
%   128-bit key and transmits the packet to the paired neighbor
within
%   range.  The receiver (RX) of an authentic advertisement will
respond
%   with an acknowledgement to the sender (TX) using the
established
%   symmetric key to ensure the key works.  The packet's payload
includes
%   the sending node's NodeID, the neighbor acknowledgement
message, and a
%   nonce.

global ackMessage
global N
global GroupKey

nonce = dec2hex(randi(2^32,1));  % generate 32-bit nonce in HEX
while length(nonce) < 8          % if nonce is less than 8 digits,
pad zeros
    nonce = cat(2,nonce,'0');
end

pad = '000000';                        % pad with zeros for AES
function

plain_text = cat(2,ackMessage,nonce,pad);

key = GroupKey;

for i = 1:length(N(RX).idStorage)
    if N(TX).nodeID == N(RX).idStorage{i}
        key = N(RX).keyStorage{i};
    end
end

%  Encrypt packet for transmission
```

```
packet = cat(2,N(RX).nodeID,nonce,aes_encrypt(plain_text,key));

*****************************************************************

function [packet] = neighborAd(nodeID,key)

%%  NEIGHBOR ADVERTISEMENT FUNCTION

%   This function encrypts a 127-octet packet using AES-128 with
the group
%   128-bit key and transmits the packet to a neighbor within
transmission
%   range.  The packet's payload includes the sending node's
NodeID, the
%   neighbor advertisement message, and a nonce.

global adMessage
global adCount

nonce = dec2hex(randi(2^32,1));  % generate 32-bit nonce in HEX
while length(nonce) < 8         % if nonce is less than 8 digits,
pad zeros
    nonce = cat(2,nonce,'0');
end

pad = '000000';                      % pad with zeros for AES
function

plain_text = cat(2,adMessage,nonce,pad);

%  Encrypt packet for transmission
packet = cat(2,nodeID,nonce,aes_encrypt(plain_text,key));

%  Increment global advertisement counter
adCount = adCount + 1;

*****************************************************************

function [NodeID] = createNodeID(nodeNumber)

NodeID = DataHash(nodeNumber);


*****************************************************************

function [ID_match] = checkNodeID(RX,TX,packet_data,option)

%%  CHECK NODE ID FUNCTION

%   This function will enable a node to read the node ID from a
packet and
%   parse its memory for a shared symmetric key.  Option 1 will
check the
%   memory of the RX, while option 2 will check the memory of the
TX.
```

```
global N

switch option
    case 1
        nodeID = packet_data(1:32);
        ID_match = 0;

        for i = 1:length(N(RX).idStorage)
            if nodeID == N(RX).idStorage{i}
                ID_match = 1;
            end
        end

    case 2
        nodeID = N(RX).nodeID;
        ID_match = 0;

        for i = 1:length(N(TX).idStorage)
            if nodeID == N(TX).idStorage{i}
                ID_match = 1;
            end
        end
end

*****************************************************************

function [cipher_text] = aes_encrypt(pt,key_string)
%% AES Encryption Function
%  This function will encrypt a plain text message using AES-128
using
%  a 128-bit key.  The inputs are the plain text string and the
key as a
%  string of hex numbers.  The function will output the cipher
text as a
%  character string.
%
%% AES Key Initialization
%  This segment receives a key as a character string and
translates it into
%  an array of 16 hex numbers, displayed in decimal.

for i=1:length(key_string)/2
    key(i) = hex2dec(key_string(i*2-1:i*2));
end

%% AES Initialization Sub-Function
%  This segment uses the AES initialization function to produce
the AES
%  structure, to include the AES parameters and tables.

s = aesinit(key);

%% AES Encryption Sub-Function

for i=1:length(pt)/2
    plain_text(i) = hex2dec(pt(i*2-1:i*2));
end
```

```
cipher_dec = aes(s,'enc','ecb',plain_text);
cipher_text = sprintf('%02x',cipher_dec);

*****************************************************************

function [plain_text] = aes_decrypt(ct,key_string)
%% AES Decryption Function
%  This function will decrypt a cipher text message with AES-128
using
%  a 128-bit key.  The inputs are the cipher text string and the
key as a
%  string of hex numbers.  The function will output the plain
text as a
%  character string.
%
%% AES Key Initialization
%  This segment receives a key as a character string and
translates it into
%  an array of 16 hex numbers, displayed in decimal.

for i=1:length(key_string)/2
    key(i) = hex2dec(key_string(i*2-1:i*2));
end

%% AES Initialization Sub-Function
%  This segment uses the AES initialization function to produce
the AES
%  structure, to include the AES parameters and tables.

s = aesinit(key);

%% AES Encryption Sub-Function

for i=1:length(ct)/2
    cipher_text(i) = hex2dec(ct(i*2-1:i*2));
end

plain_dec = aes(s,'dec','ecb',cipher_text);
plain_text = sprintf('%02x',plain_dec);


*****************************************************************

function Hash = DataHash(Data, Opt)
% DATAHASH - Checksum for Matlab array of any type
% This function creates a hash value for an input of any type.
The type and
% dimensions of the input are considered as default, such that
UINT8([0,0]) and
% UINT16(0) have different hash values. Nested STRUCTs and CELLs
are parsed
% recursively.
%
% Hash = DataHash(Data, Opt)
%
% Michael Kleder, "Compute Hash", no structs and cells:
```

```
%   http://www.mathworks.com/matlabcentral/fileexchange/8944
% Tim, "Serialize/Deserialize", converts structs and cells to a
byte stream:
%   http://www.mathworks.com/matlabcentral/fileexchange/29457

% $JRev: R-H V:033 Sum:R+m7rAPNLvlw Date:18-Jun-2016 14:33:17 $
% $License: BSD (use/copy/change/redistribute on own risk,
mention the author) $
% $File: Tools\GLFile\DataHash.m $

% Main function:
===============================================================
% Default options: ----------------------------------------------
---------------
Method    = 'MD5';
OutFormat = 'hex';
isFile    = false;
isBin     = false;

% Check number and type of inputs: ------------------------------
---------------
nArg = nargin;
if nArg == 2
   if isa(Opt, 'struct') == 0   % Bad type of 2nd input:
      Error_L('BadInput2', '2nd input [Opt] must be a struct.');
   end

   % Specify hash algorithm:
   if isfield(Opt, 'Method')  && ~isempty(Opt.Method)   % Short-
circuiting
      Method = upper(Opt.Method);
   end

   % Specify output format:
   if isfield(Opt, 'Format') && ~isempty(Opt.Format)    % Short-
circuiting
      OutFormat = Opt.Format;
   end

   % Check if the Input type is specified - default: 'array':
   if isfield(Opt, 'Input') && ~isempty(Opt.Input)      % Short-
circuiting
      if strcmpi(Opt.Input, 'File')
         if ischar(Data) == 0
            Error_L('CannotOpen', '1st input FileName must be a
string');
         end
         isFile = true;

      elseif strncmpi(Opt.Input, 'bin', 3)  % Accept 'binary'
also
         if (isnumeric(Data) || ischar(Data) || islogical(Data))
== 0 || ...
               issparse(Data)
            Error_L('BadDataType', ...
               '1st input must be numeric, CHAR or LOGICAL for
binary input.');
```

```
         end
         isBin = true;

      elseif strncmpi(Opt.Input, 'asc', 3)  % 8-bit ASCII
characters
         if ~ischar(Data)
            Error_L('BadDataType', ...
               '1st input must be a CHAR for the input type
ASCII.');
         end
         isBin = true;
         Data  = uint8(Data);
      end
   end

elseif nArg == 0  % Reply version of this function:
   R = Version_L;

   if nargout == 0
      disp(R);
   else
      Hash = R;
   end

   return;

elseif nArg ~= 1  % Bad number of arguments:
   Error_L('BadNInput', '1 or 2 inputs required.');
end

% Create the engine: -----------------------------------------
---------------
try
   Engine = java.security.MessageDigest.getInstance(Method);
catch
   Error_L('BadInput2', 'Invalid algorithm: [%s].', Method);
end

% Create the hash value: -------------------------------------
---------------
if isFile
   % Open the file:
   FID = fopen(Data, 'r');
   if FID < 0
      % Check existence of file:
      Found = FileExist_L(Data);
      if Found
         Error_L('CantOpenFile', 'Cannot open file: %s.', Data);
      else
         Error_L('FileNotFound', 'File not found: %s.', Data);
      end
   end

   % Read file in chunks to save memory and Java heap space:
   Chunk = 1e6;      % Fastest for 1e6 on Win7/64, HDD
   Count = Chunk;    % Dummy value to satisfy WHILE condition
   while Count == Chunk
```

```
      [Data, Count] = fread(FID, Chunk, '*uint8');
      if Count ~= 0  % Avoid error for empty file
         Engine.update(Data);
      end
   end
   fclose(FID);

   % Calculate the hash:
   Hash = typecast(Engine.digest, 'uint8');

elseif isBin             % Contents of an elementary array, type
tested already:
   if isempty(Data)      % Nothing to do, Engine.update fails for
empty input!
      Hash = typecast(Engine.digest, 'uint8');
   else                  % Matlab's TYPECAST is less elegant:
      if isnumeric(Data)
         if isreal(Data)
            Engine.update(typecast(Data(:), 'uint8'));
         else
            Engine.update(typecast(real(Data(:)), 'uint8'));
            Engine.update(typecast(imag(Data(:)), 'uint8'));
         end
      elseif islogical(Data)                % TYPECAST cannot
handle LOGICAL
         Engine.update(typecast(uint8(Data(:)), 'uint8'));
      elseif ischar(Data)                   % TYPECAST cannot
handle CHAR
         Engine.update(typecast(uint16(Data(:)), 'uint8'));
         % Bugfix: Line removed
      end
      Hash = typecast(Engine.digest, 'uint8');
   end
else                   % Array with type:
   Engine = CoreHash(Data, Engine);
   Hash   = typecast(Engine.digest, 'uint8');
end

% Convert hash specific output format: --------------------------
---------------
switch OutFormat
   case 'hex'
      Hash = sprintf('%.2x', double(Hash));
   case 'HEX'
      Hash = sprintf('%.2X', double(Hash));
   case 'double'
      Hash = double(reshape(Hash, 1, []));
   case 'uint8'
      Hash = reshape(Hash, 1, []);
   case 'base64'
      Hash = fBase64_enc(double(Hash));
   otherwise
      Error_L('BadOutFormat', ...
         '[Opt.Format] must be: HEX, hex, uint8, double,
base64.');
end
```

```
% return;

%
*****************************************************************
*************
function Engine = CoreHash(Data, Engine)
% This methods uses the slower TYPECAST of Matlab

% Consider the type and dimensions of the array to distinguish
arrays with the
% same data, but different shape: [0 x 0] and [0 x 1], [1,2] and
[1;2],
% DOUBLE(0) and SINGLE([0,0]):
% <  v016: [class, size, data]. BUG! 0 and zeros(1,1,0) had the
same hash!
% >= v016: [class, ndims, size, data]
Engine.update([uint8(class(Data)), ...
              typecast(uint64([ndims(Data), size(Data)]),
'uint8')]);

if issparse(Data)                     % Sparse arrays to struct:
   [S.Index1, S.Index2, S.Value] = find(Data);
   Engine                         = CoreHash(S, Engine);
elseif isstruct(Data)                 % Hash for all array
elements and fields:
   F = sort(fieldnames(Data));        % Ignore order of fields
   for iField = 1:length(F)           % Loop over fields
      aField = F{iField};
      Engine.update(uint8(aField));
      for iS = 1:numel(Data)          % Loop over elements of
struct array
         Engine = CoreHash(Data(iS).(aField), Engine);
      end
   end
elseif iscell(Data)                   % Get hash for all cell
elements:
   for iS = 1:numel(Data)
      Engine = CoreHash(Data{iS}, Engine);
   end
elseif isempty(Data)                  % Nothing to do
elseif isnumeric(Data)
   if isreal(Data)
      Engine.update(typecast(Data(:), 'uint8'));
   else
      Engine.update(typecast(real(Data(:)), 'uint8'));
      Engine.update(typecast(imag(Data(:)), 'uint8'));
   end
elseif islogical(Data)                % TYPECAST cannot handle
LOGICAL
   Engine.update(typecast(uint8(Data(:)), 'uint8'));
elseif ischar(Data)                   % TYPECAST cannot handle
CHAR
   Engine.update(typecast(uint16(Data(:)), 'uint8'));
elseif isa(Data, 'function_handle')
   Engine = CoreHash(ConvertFuncHandle(Data), Engine);
elseif (isobject(Data) || isjava(Data)) && ismethod(Data,
'hashCode')
```

```
   Engine = CoreHash(char(Data.hashCode), Engine);
else  % Most likely a user-defined object:
   try
      BasicData = ConvertObject(Data);
   catch ME
      error(['JSimon:', mfilename, ':BadDataType'], ...
         '%s: Cannot create elementary array for type: %s\n  %s',
...
        mfilename, class(Data), ME.message);
   end

   try
      Engine = CoreHash(BasicData, Engine);
   catch ME
      if strcmpi(ME.identifier, 'MATLAB:recursionLimit')
         ME = MException(['JSimon:', mfilename,
':RecursiveType'], ...
            '%s: Cannot create hash for recursive data type: %s',
...
            mfilename, class(Data));
      end
      throw(ME);
   end
end

% return;

%
*****************************************************************
*************
function FuncKey = ConvertFuncHandle(FuncH)
%   The subfunction ConvertFuncHandle converts function_handles
to a struct
%   using the Matlab function FUNCTIONS. The output of this
function changes
%   with the Matlab version, such that DataHash(@sin) replies
different hashes
%   under Matlab 6.5 and 2009a.
%   An alternative is using the function name and name of the
file for
%   function_handles, but this is not unique for nested or
anonymous functions.
%   If the MATLABROOT is removed from the file's path, at least
the hash of
%   Matlab's toolbox functions is (usually!) not influenced by
the version.
%   Finally I'm in doubt if there is a unique method to hash
function handles.
%   Please adjust the subfunction ConvertFuncHandles to your
needs.

% The Matlab version influences the conversion by FUNCTIONS:
% 1. The format of the struct replied FUNCTIONS is not fixed,
% 2. The full paths of toolbox function e.g. for @mean differ.
FuncKey = functions(FuncH);
```

```
% Include modification file time and file size. Suggested by
Aslak Grinsted:
if ~isempty(FuncKey.file)
    d = dir(FuncKey.file);
    if ~isempty(d)
        FuncKey.filebytes = d.bytes;
        FuncKey.filedate  = d.datenum;
    end
end

% ALTERNATIVE: Use name and path. The <matlabroot> part of the
toolbox functions
% is replaced such that the hash for @mean does not depend on the
Matlab
% version.
% Drawbacks: Anonymous functions, nested functions...
% funcStruct = functions(FuncH);
% funcfile   = strrep(funcStruct.file, matlabroot, '<MATLAB>');
% FuncKey    = uint8([funcStruct.function, ' ', funcfile]);

% Finally I'm afraid there is no unique method to get a hash for
a function
% handle. Please adjust this conversion to your needs.

% return;

%
*****************************************************************
*************
function DataBin = ConvertObject(DataObj)
% Convert a user-defined object to a binary stream. There cannot
be a unique
% solution, so this part is left for the user...

try    % Perhaps a direct conversion is implemented:
   DataBin = uint8(DataObj);

   % Matt Raum had this excellent idea - unfortunately this
function is
   % undocumented and might not be supported in te future:
   % DataBin = getByteStreamFromArray(DataObj);

catch  % Or perhaps this is better:
   WarnS   = warning('off', 'MATLAB:structOnObject');
   DataBin = struct(DataObj);
   warning(WarnS);
end

% return;

%
*****************************************************************
*************
function Out = fBase64_enc(In)
% Encode numeric vector of UINT8 values to base64 string.
% The intention of this is to create a shorter hash than the HEX
format.
```

```
% Therefore a padding with '=' characters is omitted on purpose.

Pool = [65:90, 97:122, 48:57, 43, 47];  % [0:9, a:z, A:Z, +, /]
v8   = [128; 64; 32; 16; 8; 4; 2; 1];
v6   = [32, 16, 8, 4, 2, 1];

In  = reshape(In, 1, []);
X   = rem(floor(In(ones(8, 1), :) ./ v8(:, ones(length(In), 1))),
2);
Y   = reshape([X(:); zeros(6 - rem(numel(X), 6), 1)], 6, []);
Out = char(Pool(1 + v6 * Y));

% return;

%
*****************************************************************
*************
function Ex = FileExist_L(FileName)
% A more reliable version of EXIST(FileName, 'file'):
dirFile = dir(FileName);
if length(dirFile) == 1
   Ex = ~(dirFile.isdir);
else
   Ex = false;
end

% return;

%
*****************************************************************
*************
function R = Version_L()
% The output differs between versions of this function. So give
the user a
% chance to recognize the version:
% 1: 01-May-2011, Initial version
% 2: 15-Feb-2015, The number of dimensions is considered in
addition.
%    In version 1 these variables had the same hash:
%    zeros(1,1) and zeros(1,1,0), complex(0) and zeros(1,1,0,0)
% 3: 29-Jun-2015, Struct arrays are processed field by field and
not element
%    by element, because this is much faster. In consequence the
hash value
%    differs, if the input contains a struct.
% 4: 28-Feb-2016 15:20, same output as GetMD5 for MD5 sums.
Therefore the
%    dimensions are casted to UINT64 at first.
R.HashVersion = 4;
R.Date        = [2016, 2, 28];

R.HashMethod  = {};
try
   Provider = java.security.Security.getProviders;
   for iProvider = 1:numel(Provider)
      S     = char(Provider(iProvider).getServices);
      Index = strfind(S, 'MessageDigest.');
      for iDigest = 1:length(Index)
         Digest       = strtok(S(Index(iDigest):end));
         Digest       = strrep(Digest, 'MessageDigest.', '');
         R.HashMethod = cat(2, R.HashMethod, {Digest});
      end
   end
catch ME
   fprintf(2, '%s\n', ME.message);
   R.HashMethod = 'error';
end

% return;

%
*****************************************************************
*************
function Error_L(ID, varargin)

error(['JSimon:', mfilename, ':', ID], ['*** %s: ', varargin{1}],
...
   mfilename, varargin{2:nargin - 1});

% return;


*****************************************************************

function [output] = aes(s, oper, mode, input, iv, sbit)
% AES Encrypt/decrypt array of bytes by AES.
% output = aes(s, oper, mode, input, iv, sbit)
% Encrypt/decrypt array of bytes by AES-128, AES-192, AES-256.
% All NIST SP800-38A cipher modes supported (e.g. ECB, CBC, OFB,
CFB, CTR).
% Usage example:    out = aesdecrypt(s, 'dec', 'ecb', data)
% s:                AES structure (generated by aesinit)
% oper:             operation:
%                   'e', 'enc', 'encrypt', 'E',... = encrypt
%                   'd', 'dec', 'decrypt', 'D',... = decrypt
% mode:             operation mode
%                   'ecb' = Electronic Codebook Mode
%                   'cbc' = Cipher Block Chaining Mode
%                   'cfb' = Cipher Feedback Mode
%                   'ofb' = Output Feedback Mode
%                   'ctr' = Counter Mode
%                   For counter mode you need external
AES_GET_COUNTER()
%                   counter function.
% input:            plaintext/ciphertext byte-vector with length
%                   multiple of 16
% iv:               initialize vector - some modes need it
%                   ending initialize vector is stored in s.iv,
so you
%                   can use aes() repetitively to encode/decode
%                   large vector:
%                   out = aes(s, 'enc', 'cbc', input1, iv);
%                   out = [out aes(s, 'enc', 'cbc', input1,
s.iv)];
```

```
%                    ...
% sbit:              bit-width parameter for CFB mode
% output:            ciphertext/plaintext byte-vector
%
% See
% Morris Dworkin, Recommendation for Block Cipher Modes of
Operation
% Methods and Techniques
% NIST Special Publication 800-38A, 2001 Edition
% for details.

% Stepan Matejka, 2011, matejka[at]feld.cvut.cz
% $Revision: 1.1.0 $  $Date: 2011/10/12 $

error(nargchk(4, 6, nargin));

validateattributes(s, {'struct'}, {});
validateattributes(oper, {'char'}, {});
validateattributes(mode, {'char'}, {});
validateattributes(input, {'numeric'}, {'real', 'vector', '>=',
0, '<', 256});
if (nargin >= 5)
    validateattributes(iv, {'numeric'}, {'real', 'vector', '>=',
0, '<', 256});
    if (length(iv) ~= 16)
        error('Length of ''iv'' must be 16.');
    end
end
if (nargin >= 6)
    validateattributes(sbit, {'numeric'}, {'real', 'scalar',
'>=', 1, '<=', 128});
end

if (mod(length(input), 16))
    error('Length of ''input'' must be multiple of 16.');
end

switch lower(oper)
    case {'encrypt', 'enc', 'e'}
        oper = 0;
    case {'decrypt', 'dec', 'd'}
        oper = 1;
    otherwise
        error('Bad ''oper'' parameter.');
end

blocks = length(input)/16;
input = input(:);

switch lower(mode)

    case {'ecb'}
        % Electronic Codebook Mode
        % ------------------------
        output = zeros(1,length(input));
        idx = 1:16;
        for i = 1:blocks
```

```
            if (oper)
                % decrypt
                output(idx) = aesdecrypt(s,input(idx));
            else
                % encrypt
                output(idx) = aesencrypt(s,input(idx));
            end
            idx = idx + 16;
        end

    case {'cbc'}
        % Cipher Block Chaining Mode
        % --------------------------
        if (nargin < 5)
            error('Missing initialization vector ''iv''.');
        end
        output = zeros(1,length(input));
        ob = iv;
        idx = 1:16;
        for i = 1:blocks
            if (oper)
                % decrypt
                in = input(idx);
                output(idx) = bitxor(ob(:), aesdecrypt(s,in)');
                ob = in;
            else
                % encrypt
                ob = bitxor(ob(:), input(idx));
                ob = aesencrypt(s, ob);
                output(idx) = ob;
            end
            idx = idx + 16;
        end
        % store iv for block passing
        s.iv = ob;

    case {'cfb'}
        % Cipher Feedback Mode
        % --------------------
        % Special mode with bit manipulations
        % sbit = 1..128
        if (nargin < 6)
            error('Missing ''sbit'' parameter.');
        end
        % get number of bits
        bitlen = 8*length(input);
        % loop counter
        rounds = round(bitlen/sbit);
        % check
        if (rem(bitlen, sbit))
            error('Message length in bits is not multiple of
''sbit''.');
        end
        % convert input to bitstream
        inputb = reshape(de2bi(input,8,2,'left-msb')',1,bitlen);
        % preset init. vector
        ib = iv;
        ibb = reshape(de2bi(ib,8,2,'left-msb')',1,128);
        % preset output binary stream
        outputb = zeros(size(inputb));
        for i = 1:rounds
            iba = aesencrypt(s, ib);
            % convert to bit, MSB first
            ibab = reshape(de2bi(iba,8,2,'left-msb')',1,128);
            % strip only sbit MSB bits
            % this goes to xor
            ibab = ibab(1:sbit);
            % strip bits from input
            inpb = inputb((i - 1)*sbit + (1:sbit));
            % make xor
            outb = bitxor(ibab, inpb);
            % write to output
            outputb((i - 1)*sbit + (1:sbit)) = outb;
            if (oper)
                % decrypt
                % prepare new iv - bit shift
                ibb = [ibb((1 + sbit):end) inpb];
            else
                % encrypt
                % prepare new iv - bit shift
                ibb = [ibb((1 + sbit):end) outb];
            end
            % back to byte ary
            ib = bi2de(vec2mat(ibb,8),'left-msb');
            % loop
        end
        output = bi2de(vec2mat(outputb,8),'left-msb');
        % store iv for block passing
        s.iv = ib;

    case {'ofb'}
        % Output Feedback Mode
        % --------------------
        if (nargin < 5)
            error('Missing initialization vector ''iv''.');
        end
        output = zeros(1,length(input));
        ib = iv;
        idx = 1:16;
        for i = 1:blocks
            % encrypt, decrypt
            ib = aesencrypt(s, ib);
            output(idx) = bitxor(ib(:), input(idx));
            idx = idx + 16;
        end
        % store iv for block passing
        s.iv = ib;

    case {'ctr'}
        % Counter Mode
        % ------------
        if (nargin < 5)
            iv = 1;
        end
        output = zeros(1,length(input));
        idx = 1:16;
        for i = (iv):(iv + blocks - 1)
            ib = AES_GET_COUNTER(i);
            ib = aesencrypt(s, ib);
            output(idx) = bitxor(ib(:), input(idx));
            idx = idx + 16;
        end
        s.iv = iv + blocks;

    %otherwise
        %error('Bad ''oper'' parameter.');
end

% -----------------------------------------------------------------
---------
% end of file

*****************************************************************

function s = aesinit(key)
% AESINIT Generate structure with s-boxes, expanded key, etc.
% Usage:            s = aesinit([23 34 168 ... 39])
% key:              16 (AES-128), 24 (AES-192), and 32 (AES-256)
%                   items array with bytes of key
% s:                AES structure for AES parameters and tables

% Stepan Matejka, 2011, matejka[at]feld.cvut.cz
% $Revision: 1.1.0 $  $Date: 2011/10/12 $

validateattributes(key,...
    {'numeric'},...
    {'real', 'vector', '>=', 0, '<=', 255});

key = key(:);
lengthkey = length(key);

switch (lengthkey)
    case 16
        rounds = 10;
    case 24
        rounds = 12;
    case 32
        rounds = 14;
    otherwise
        error('Only AES-128, AES-192, and AES-256 are
supported.');
end

% fill s structure
s = {};
s.key = key;
s.bytes = lengthkey;
s.length = lengthkey * 8;
s.rounds = rounds;
% irreducible polynomial for multiplication in a finite field
0x11b
```

```
% bin2dec('100011011');
s.mod_pol = 283;

% s-box method 2 (faster)
% ------------------------

% first build logarithm lookup table and it's inverse
aes_logt = zeros(1,256);
aes_ilogt = zeros(1,256);
gen = 1;
for i = 0:255
    aes_logt(gen + 1) = i;
    aes_ilogt(i + 1) = gen;
    gen = poly_mult(gen, 3, s.mod_pol);
end
% store log tables
s.aes_logt = aes_logt;
s.aes_ilogt = aes_ilogt;
% build s-box and it's inverse
s_box = zeros(1,256);
loctable = [1 2 4 8 16 32 64 128 1 2 4 8 16 32 64 128];
for i = 0:255
    if (i == 0)
        inv = 0;
    else
        inv = aes_ilogt(255 - aes_logt(i + 1) + 1);
    end
    temp = 0;
    for bi = 0:7
        temp2 = sign(bitand(inv, loctable(bi + 1)));
        temp2 = temp2 + sign(bitand(inv, loctable(bi + 4 + 1)));
        temp2 = temp2 + sign(bitand(inv, loctable(bi + 5 + 1)));
        temp2 = temp2 + sign(bitand(inv, loctable(bi + 6 + 1)));
        temp2 = temp2 + sign(bitand(inv, loctable(bi + 7 + 1)));
        temp2 = temp2 + sign(bitand(99, loctable(bi + 1)));
        if (rem(temp2,2))
            temp = bitor(temp, loctable(bi + 1));
        end
    end
    s_box(i + 1) = temp;
end
inv_s_box(s_box(1:256) + 1) = (0:255);
% table correction (must be)
s_box(1 + 1) = 124;
inv_s_box(124 + 1) = 1;
inv_s_box(99 + 1) = 0;
s.s_box = s_box;
s.inv_s_box = inv_s_box;

% tables for fast MixColumns
mix_col2 = zeros(1,256);
mix_col3 = mix_col2;
mix_col9 = mix_col2;
mix_col11 = mix_col2;
mix_col13 = mix_col2;
mix_col14 = mix_col2;
for i = 1:256
```

```
    mix_col2(i) = poly_mult(2, i - 1, s.mod_pol);
    mix_col3(i) = poly_mult(3, i - 1, s.mod_pol);
    mix_col9(i) = poly_mult(9, i - 1, s.mod_pol);
    mix_col11(i) = poly_mult(11, i - 1, s.mod_pol);
    mix_col13(i) = poly_mult(13, i - 1, s.mod_pol);
    mix_col14(i) = poly_mult(14, i - 1, s.mod_pol);
end
s.mix_col2 = mix_col2;
s.mix_col3 = mix_col3;
s.mix_col9 = mix_col9;
s.mix_col11 = mix_col11;
s.mix_col13 = mix_col13;
s.mix_col14 = mix_col14;

% expanded key
s.keyexp = key_expansion(s.key, s.s_box, s.rounds, s.mod_pol,
s.aes_logt, s.aes_ilogt);

% poly & invpoly
s.poly_mat = [...
    2 3 1 1;...
    1 2 3 1;...
    1 1 2 3;...
    3 1 1 2];

s.inv_poly_mat =[...
    14 11 13  9;...
    9  14 11 13;...
    13  9 14 11;...
    11 13  9 14];

% end of aesinit.m
% ------------------------------------------------------------------
---------

function p = poly_mult(a, b, mod_pol)
% Multiplication in a finite field
% For loop multiplication - slower than log/ilog tables
% but must be used for log/ilog tables generation

p = 0;
for counter = 1 : 8
    if (rem(b, 2))
        p = bitxor(p, a);
        b = (b - 1)/2;
    else
        b = b/2;
    end
    a = 2*a;
    if (a > 255)
        a = bitxor(a, mod_pol);
    end
end

% ------------------------------------------------------------------
---------
function inv = find_inverse(in, mod_pol)
```

```
% Multiplicative inverse for an element a of a finite field
% very bad calculate & test method
% Not used in faster version

% loop over all possible bytes
for inv = 1 : 255
    % calculate polynomial multiplication and test to be 1
    if (1 == poly_mult(in, inv, mod_pol))
        % we find it
        break
    end
end
inv = 0;

% ------------------------------------------------------------------
---------
function out = aff_trans(in)
% Affine transformation over GF(2^8)
% Not used for faster s-box generation

% modulo polynomial for multiplication in a finite field
% bin2dec('100000001');
mod_pol = 257;

% multiplication polynomial
% bin2dec('00011111');
mult_pol = 31;

% addition polynomial
% bin2dec('01100011');
add_pol = 99;

% polynomial multiplication
temp = poly_mult(in, mult_pol, mod_pol);

% xor with addition polynomial
out = bitxor(temp, add_pol);

% ------------------------------------------------------------------
---------
function expkey = key_expansion(key, s_box, rounds, mod_pol,
aes_logt, aes_ilogt)
% Expansion of key

% This is new faster version for all AES:
rcon = 1;
kcol = length(key)/4;
expkey = (reshape(key,4,kcol))';
% traverse for all rounds
for i = kcol:(4*(rounds + 1) - 1)
    % copy the previous row of the expanded key into a buffer
    temp = expkey(i, :);
    % each kcol row
    if (mod(i, kcol) == 0)
        % rotate word
        temp = temp([2 3 4 1]);
        % s-box transform
        temp = s_box(temp + 1);
        % xor
        temp(1) = bitxor(temp(1), rcon);
        % new rcon
        % 1. classic poly_mult
        % rcon = poly_mult(rcon, 2, mod_pol);
        % 2. or faster version with log/ilog tables
        % note rcon is never zero here
        % rcon = aes_ilogt(mod((aes_logt(rcon + 1) + aes_logt(2 +
1)), 255) + 1);
        rcon = aes_ilogt(mod((aes_logt(rcon + 1) + 25), 255) +
1);
    else
        if ((kcol > 6) && (mod(i, kcol) == 4))
            temp = s_box(temp + 1);
        end
    end
    % generate new row of the expanded key
    expkey(i + 1, :) = bitxor(expkey(i - kcol + 1, :), temp);
end

% -----------------------------------------------------------------
----------
% end of file

*****************************************************************

function [out] = aesencrypt(s, in)
% AESENCRYPT  Encrypt 16-bytes vector.
% Usage:            out = aesencrypt(s, in)
% s:                AES structure
% in:               input 16-bytes vector (plaintext)
% out:              output 16-bytes vector (ciphertext)

% Stepan Matejka, 2011, matejka[at]feld.cvut.cz
% $Revision: 1.1.0 $  $Date: 2011/10/12 $

if (nargin ~= 2)
    error('Bad number of input arguments.');
end

validateattributes(s, {'struct'}, {});
validateattributes(in, {'numeric'}, {'real', 'vector', '>=', 0,
'<', 256});

% copy input to local
% 16 -> 4 x 4
state = reshape(in, 4, 4);

% Initial round
% AddRoundKey keyexp(1:4)
state = bitxor(state, (s.keyexp(1:4, :))');

% Loop over (s.rounds - 1) rounds
for i = 1:(s.rounds - 1)
    % SubBytes - lookup table
    state = s.s_box(state + 1);
    % ShiftRows
    state = shift_rows(state, 0);
    % MixColumns
    state = mix_columns(state, s);
    % AddRoundKey keyexp(i*4 + (1:4))
    state = bitxor(state, (s.keyexp((1:4) + 4*i, :))');
end

% Final round
% SubBytes - lookup table
state = s.s_box(state + 1);
% ShiftRows
state = shift_rows(state, 0);
% AddRoundKey keyexp(4*s.rounds + (1:4))
state = bitxor(state, (s.keyexp(4*s.rounds + (1:4), :))');

% copy local to output
% 4 x 4 -> 16
out = reshape(state, 1, 16);

% ---------------------------------------------------------------
---------
function out = mix_columns(in, s)
% Each column of the state is multiplied with a fixed polynomial
mod_pol

% Faster faster faster faster implementation
out = bitxor(bitxor(bitxor([in(3,1:4); in(1,1:4); in(1,1:4);
in(2,1:4)],...
    [in(4,1:4); in(4,1:4); in(2,1:4); in(3,1:4)]),...
    [s.mix_col2(in(1,1:4) + 1); s.mix_col2(in(2,1:4) + 1);
s.mix_col2(in(3,1:4) + 1); s.mix_col3(in(1,1:4) + 1)]),...
    [s.mix_col3(in(2,1:4) + 1); s.mix_col3(in(3,1:4) + 1);
s.mix_col3(in(4,1:4) + 1); s.mix_col2(in(4,1:4) + 1)]);

% ---------------------------------------------------------------
---------
function p = poly_mult(a, b, mod_pol, aes_logt, aes_ilogt)
% Multiplication in a finite field

% Faster implementaion
if (a && b)
    p = aes_ilogt(mod((aes_logt(a + 1) + aes_logt(b + 1)), 255) +
1);
else
    p = 0;
end

% ---------------------------------------------------------------
---------
function out = shift_rows(in, dir)
% ShiftRows cyclically shift the rows of the 4 x 4 matrix.
%
%   dir = 0 (to left)
%   | 1 2 3 4 |
%   | 2 3 4 1 |
%   | 3 4 1 2 |
%   | 4 1 2 3 |
%
%    dir ~= 0 (to right)
%   | 1 2 3 4 |
%   | 4 1 2 3 |
%   | 3 4 1 2 |
%   | 2 3 4 1 |
%

if (dir == 0)
    % left
    % use linear indexing in 2d array
    out = reshape(in([1 6 11 16 5 10 15 4 9 14 3 8 13 2 7
12]),4,4);
    % old safe method
%     temp = reshape(in,16,1);
%     temp = temp([1 6 11 16 5 10 15 4 9 14 3 8 13 2 7 12]);
%     out = reshape(temp,4,4);
else
    % right
    % use linear indexing in 2d array
    out = reshape(in([1 14 11 8 5 2 15 12 9 6 3 16 13 10 7
4]),4,4);
    % old safe method
%     temp = reshape(in,16,1);
%     temp = temp([1 14 11 8 5 2 15 12 9 6 3 16 13 10 7 4]);
%     out = reshape(temp,4,4);
end

% ------------------------------------------------------------------
---------
% end of file

*******************************************************************

function [out] = aesdecrypt(s, in)
% AESDECRYPT Decrypt 16-bytes vector.
% Usage:            out = aesdecrypt(s, in)
% s:                AES structure
% in:               input 16-bytes vector (ciphertext)
% out:              output 16-bytes vector (plaintext)

% Stepan Matejka, 2011, matejka[at]feld.cvut.cz
% $Revision: 1.1.0 $  $Date: 2011/10/12 $

if (nargin ~= 2)
    error('Bad number of input arguments.');
end

validateattributes(s, {'struct'}, {});
validateattributes(in, {'numeric'}, {'real', 'vector', '>=', 0,
'<', 256});

% copy input to local
% 16 -> 4 x 4
state = reshape(in, 4, 4);
```

```
% Initial round
% AddRoundKey keyexp(s.rounds*4 + (1:4))
state = bitxor(state, (s.keyexp(s.rounds*4 + (1:4), :))');

% Loop over (s.rounds - 1) rounds
for i = (s.rounds - 1):-1:1
    % ShiftRows
    state = shift_rows(state, 1);
    % SubBytes - lookup table
    state = s.inv_s_box(state + 1);
    % AddRoundKey keyexp(i*4 + (1:4))
    state = bitxor(state, (s.keyexp((1:4) + 4*i, :))');
    % MixColumns
    state = mix_columns(state, s);
end

% Final round
% ShiftRows
state = shift_rows(state, 1);
% SubBytes - lookup table
state = s.inv_s_box(state + 1);
% AddRoundKey keyexp(1:4)
state = bitxor(state, (s.keyexp(1:4, :))');

% copy local to output
% 4 x 4 -> 16
out = reshape(state, 1, 16);

% ---------------------------------------------------------------
---------
function out = mix_columns(in, s)
% Each column of the state is multiplied with a fixed polynomial
mod_pol

% Faster faster faster faster implementation
out = bitxor(bitxor(bitxor(...
    [s.mix_col14(in(1,1:4) + 1); s.mix_col9(in(1,1:4) +
1);  s.mix_col13(in(1,1:4) + 1); s.mix_col11(in(1,1:4) + 1)],...
    [s.mix_col11(in(2,1:4) + 1); s.mix_col14(in(2,1:4) + 1);
s.mix_col9(in(2,1:4) + 1);  s.mix_col13(in(2,1:4) + 1)]),...
    [s.mix_col13(in(3,1:4) + 1); s.mix_col11(in(3,1:4) + 1);
s.mix_col14(in(3,1:4) + 1); s.mix_col9(in(3,1:4) + 1)]),...
    [s.mix_col9(in(4,1:4) + 1);  s.mix_col13(in(4,1:4) + 1);
s.mix_col11(in(4,1:4) + 1); s.mix_col14(in(4,1:4) + 1)]);

% ---------------------------------------------------------------
---------
function p = poly_mult(a, b, mod_pol, aes_logt, aes_ilogt)
% Multiplication in a finite field

% Old slow implementation
% p = 0;
% for counter = 1:8
%     if (rem(b,2))
%         p = bitxor(p,a);
%         b = (b - 1)/2;
%     else
%         b = b/2;
%     end
%     a = 2*a;
%     if (a>255)
%         a = bitxor(a,mod_pol);
%     end
% end

% Faster implementaion
if (a && b)
    p = aes_ilogt(mod((aes_logt(a + 1) + aes_logt(b + 1)), 255) +
1);
else
    p = 0;
end

% ------------------------------------------------------------------
---------
function out = shift_rows(in, dir)
% ShiftRows cyclically shift the rows of the 4 x 4 matrix.
%
%   dir = 0 (to left)
%   | 1 2 3 4 |
%   | 2 3 4 1 |
%   | 3 4 1 2 |
%   | 4 1 2 3 |
%
%   dir ~= 0 (to right)
%   | 1 2 3 4 |
%   | 4 1 2 3 |
%   | 3 4 1 2 |
%   | 2 3 4 1 |
%

if (dir == 0)
    % left
    % use linear indexing in 2d array
    out = reshape(in([1 6 11 16 5 10 15 4 9 14 3 8 13 2 7
12]),4,4);
    % old safe method
%     temp = reshape(in,16,1);
%     temp = temp([1 6 11 16 5 10 15 4 9 14 3 8 13 2 7 12]);
%     out = reshape(temp,4,4);
else
    % right
    % use linear indexing in 2d array
    out = reshape(in([1 14 11 8 5 2 15 12 9 6 3 16 13 10 7
4]),4,4);
    % old safe method
%     temp = reshape(in,16,1);
%     temp = temp([1 14 11 8 5 2 15 12 9 6 3 16 13 10 7 4]);
%     out = reshape(temp,4,4);
end

% ------------------------------------------------------------------
---------
% end of file
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] *The Marine Corps Operating Concept: How an Expeditionary Force Operates in the 21st Century*, Washington, DC: United States Marine Corps, 2016.

[2] *Marine Corps Reference Publication 2–10A.5, Remote Sensor Operations*, Washington, DC: United States Marine Corps, 2016.

[3] M. P. Đurišić, Z. Tafa, G. Dimić and V. Milutinović, "A survey of military applications of wireless sensor networks," in *Proc. of Mediterranean Conference on Embedded Computing*, Bar, Montenegro, 2012, pp. 196–199.

[4] S. H. Yang, *Wireless Sensor Networks, Principles, Design, and Applications*, London, UK: Springer-Verlag, 2014.

[5] *IEEE Standard for Local and Metropolitan Area Networks—Part 15.4: Low-Rate Wireless Personal Area Networks*, IEEE Standard 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011), 2015.

[6] *Advanced Encryption Standard*, Federal Information Processing Standards Publication 197, 2001.

[7] *Security Requirements for Cryptographic Modules*, Federal Information Processing Standards Publication 140–2, 2001.

[8] *Internet Protocol, Version 6 (IPv6) Specification*, Request for Comments 2460, 1998.

[9] *Transmission of IPv6 Packets over IEEE 802.15.4 Networks*, Request for Comments 4944, 2007.

[10] A. Gunawan. (2013, May 27). M2M optimizations in public mobile networks. [Online]. Available: https://www.slideshare.net/hamdani2/m2m-day-two.

[11] *Updated Marine Corps Policy for Use of Public Key Infrastructure (PKI) Certificates on Portable Electronic Devices (PEDS) Security and Application of Email Signature and Encryption Policy*, MARADMIN 367/17, United States Marine Corps, Washington, DC, 2017.

[12] W. Stallings, *Data and Computer Communications* Eighth Edition, Upper Saddle River, NJ: Pearson Prentice Hall, 2007.

[13] K. A. Shim, "A Survey of Public-Key Cryptographic Primitives in Wireless Sensor Networks," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 577–601, First Quarter 2016.

[14] *Neighbor Discovery for IPv6*, Request for Comments 2461, 1998.

[15] *SEcure Neighbor Discovery (SEND)*, Request for Comments 3971, 2005.

[16] J. Granjal, E. Monteiro and J. Sá Silva, "Security for the Internet of Things: A Survey of Existing Protocols and Open Research Issues," in *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, pp. 1294–1312, Third Quarter 2015.

[17] *Cryptographically Generated Addresses (CGA)*, Request for Comments 3972, 2005.

[18] A. AlSa'deh and C. Meinel, "Secure neighbor discovery: Review, challenges, perspectives, and recommendations," in *IEEE Security & Privacy*, vol. 10, no. 4, pp. 26–34, July-Aug. 2012.

[19] N. R. Potlapally, S. Ravi, A. Raghunathan and N. K. Jha, "A study of the energy consumption characteristics of cryptographic algorithms and security protocols," in *IEEE Transactions on Mobile Computing*, vol. 5, no. 2, pp. 128–143, Feb. 2006.

[20] A. S. Wander, N. Gura, H. Eberle, V. Gupta and S. C. Shantz, "Energy analysis of public-key cryptography for wireless sensor networks," *Third IEEE International Conference on Pervasive Computing and Communications*, 2005, pp. 324–328.

[21] P. Porambage, A. Braeken, C. Schmitt, A. Gurtov, M. Ylianttila and B. Stiller, "Group key establishment for enabling secure multicast communication in wireless sensor networks deployed for IoT applications," in *IEEE Access*, vol. 3, pp. 1503–1511, 2015.

[22] L. M. L. Oliveira, J. J. P. C. Rodrigues, A. F. de Sousa and V. M. Denisov, "Network admission control solution for 6lowpan networks based on symmetric key mechanisms," in *IEEE Transactions on Industrial Informatics*, vol. 12, no. 6, pp. 2186–2195, Dec. 2016.

[23] A. Shamir, "Identity-based cryptosystems and signature schemes," in *Proc. Crypto*, 1984, pp. 47–53.

[24] P. Thulasiraman, EC 4770, Class Lecture, Topic: "Diffie-Hellman and RSA." Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA, Winter 2016.

[25] W. Diffie and M. Hellman, "New directions in cryptography," in *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov 1976.

[26] T. J. Haakensen, “Achieving sink node anonymity in tactical wireless sensor networks using a reactive routing protocol,” M.S. thesis, Dept. Elec. Eng., Naval Postgraduate School, Monterey, CA, 2017.

[27] M. Siekkinen, M. Hiienkari, J. K. Nurminen, and J. Nieminen, “How low energy is Bluetooth low energy? Comparative measures with ZigBee/802.15.4,” in *Proc. of IEEE WCNC Workshop on Internet of Things Enabling Technologies, Embracing Machine-To-Machine Communications and Beyond*, 2012, pp. 232–237.

[28] S. Matejka, Prague, Czech Republic. (2011). *AES-128, AES-192, and AES-256 encryption/decryption functions*. [Online]. Available: http://radio.feld.cvut.cz/personal/matejka/wiki/doku.php?id=root:en:projects. Accessed Aug. 8, 2017.

[29] J. Simon, Heidelberg, Germany. (2016). *DataHash – Hash for Matlab array, struct, cell or file*. [Online]. Available: http://www.mathworks.com/matlabcentral/fileexchange/31272-datahash. Accessed Aug. 8, 2017.

[30] S. Adibi, Mobile Health: A Technology Road Map, Switzerland, Springer International Publishing, 2015.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California